DataGrid

Pan Language Specification

version 2.0.2

Document identifier:	**DataGrid-04-TED-0153**
EDMS id:	**357584**
Date:	**April 2, 2003**
Work package:	**Fabric Management (WP4)**
Partner(s):	**CERN**
Lead Partner:	**CERN**
Document status:	**APPROVED**
Author(s):	Lionel Cons and Piotr Poznański
File:	**pan-spec**

Abstract: *This document describes Pan, the High Level Description (HLD) language used to describe system configurations for the EDG Fabric Management Work Package (WP4).*

CONTENTS

1. INTRODUCTION

This document describes Pan, a High Level Description (HLD) language used to represent system configurations. The HLD source files reside in the Configuration Database (CDB)[1] and are compiled into Low Level configuration Descriptions (LLD). The LLD specification is available in an other document[3].

1.1. DOCUMENT CONVENTIONS

Here are the typographical conventions used in this document.

All the examples look like this:

```
example ...
```

In paragraphs, configuration element paths are represented like this: *"/hardware/cpus"*.

Formal syntax descriptions are represented inside boxes, for instance:

statement $\mapsto$ **include** *template-name* '**;**'

Non-terminal symbols (such as *statement*) are represented in italic. Terminal symbols (such as **include**) are represented in bold. When the font face is not sufficient to recognise them, terminal symbols are enclosed between quotes (such as '**;**').

Grouping is done through parenthesis, alternation is represented by a vertical bar (i.e. |) and optional parts with square brackets.

1.2. CONFIGURATION INFORMATION

The configuration information is represented by a tree of configuration parameters. For instance:

```
/hardware/memory/size = 256
/hardware/cpus/0/vendor = GenuineIntel
/hardware/cpus/0/model = Pentium III (Coppermine)
/hardware/cpus/0/speed = 800
/system/filesystems/0/name = root
/system/filesystems/0/device = /dev/hda1
/system/filesystems/0/mountpoint = /
/system/filesystems/0/type = ext2
/system/filesystems/0/options = defaults
/system/filesystems/1/name = cd
/system/filesystems/1/device = /dev/cdrom
/system/filesystems/1/mountpoint = /mnt/cdrom
/system/filesystems/1/type = iso9660
/system/filesystems/1/options = noauto,owner,ro
```

Simple values (like strings or numbers) form the leaves of this tree and are called *properties*. Internal nodes of the tree are called *resources* and are used to group *elements*[1] into *lists* (accessed by index) or named lists (aka *nlists*, accessed by name). Nlists can conveniently be used to represent tables or records[2]. Every element has a unique *path* which identifies its position in the tree.

1.3. PATHS

In Pan source files, paths are represented by strings with a format similar to UNC.

A path can be *relative* (does not start with a slash and is relative to wherever it will be hooked in the tree, e.g. `"speed"`), *absolute* (starts with a single slash and uniquely identifies a configuration parameter, e.g. `"/hardware/memory/size"`) or *external* (starts with a double slash and contains the name of the object it refers to followed by an absolute path, e.g. `"//pcsrv05/hardware/memory/size"`).

Path components can only be identifiers (see section 1.4.) or numbers. A component such as `cache-size` or `1x` are therefore invalid.

A numerical component is always interpreted as a child index inside a list. Therefore, a path such as `"/hardware/cpus/0"` identifies the first element of the list `"/hardware/cpus"`. Like in C, indexes always start at 0.

Non-numerical components identify named children within nlists. Decoding `"/hardware/cpus"`, we see that `"/"` is a nlist containing a child named ”hardware” and `"/hardware"` is also a nlist with a child named ”cpus”.

1.4. IDENTIFIERS AND NAMES

A valid *identifier* is made of one letter or underscore (‘_’) character followed by zero or more letters, digits or underscore characters. Therefore `x1` and `foo_bar` are valid while `1x`, `foo-bar` and `foo::bar` are not.

A valid *template-name* is made of one letter, digit or underscore character followed by zero or more letters, digits or underscore, minus (‘−’), plus (‘+’) or dot (‘.’) characters. Any valid Internet host name (e.g. `127.0.0.1` or `myhost.foo-bar.org`) is therefore a valid template name.

In Pan, all the other names (of variables, functions, types. . .) must be valid identifiers. Also, they cannot be reserved keywords (i.e. the terminal symbols in the syntax definitions) such as **include** or **type**.

It is allowed to use the same name for different things. You can have at the same time a function named `foo` and a variable named `foo`.

1.5. PROCESSING

The exact behaviour of the compiler (including its command line options, error messages, . . .) is not part of this specification and should be described in the compiler’s documentation.

However, for each template, the processing of the Pan sources must happen in three consecutive stages. First, during the *compilation*, all the Pan statements are executed and the configuration tree in memory is updated accordingly. Secondly, during *validation*, the (now read-only) configuration tree is

[1]The term *element* refers to either a property or a resource.
[2]I.e. similar to Pascal’s `record` or C’s `struct`.

validated using type and validation information. Finally, after a successful validation, the output file (LLD) is created.

2. MAIN SYNTAX

Pan source files can include comments preceded by the hash sign (i.e. '#'). Everything from the hash sign to the end of the line will be ignored. Space is not significant except of course when used inside strings.

Pan consists of statements grouped into templates that are in turn grouped into source files. A source file must contain at least one template and a template at least one statement. A template starts with a template declaration and contains everything up to the end of the file or the next template declaration. Formally:

file $\mapsto$ *template-seq*
template-seq $\mapsto$ *template* [*template-seq*]
template $\mapsto$ *template-decl* *statement-seq*
statement-seq $\mapsto$ *statement* [*statement-seq*]

2.1. TEMPLATES

There are four types of templates with slightly different behaviours. The type of template is specified in the template declaration:

template-decl $\mapsto$ [**declaration** | **object** | **structure**] **template** *template-name* ';'

Templates are identified by unique names (the *template-name* above) and it is recommended to use a naming convention to logically group similar templates. For instance, all the structure templates representing disks could start with `disk_`, like in Appendix C. This naming convention is outside of the scope of this document.

An *ordinary template* (i.e. without any leading keyword) can contain any kind of statement but assignments and delete statements must act on absolute paths.

A *declaration template* can only contain declaration statements, i.e. that do not modify the configuration tree[3]. Also, it can only include other declaration templates. When included several times, the declaration statements are executed only once. This is similar to C header files with the magic `#ifdef` lines preventing multiple (re)declarations.

An *object template* behaves like an ordinary template except that it is associated with a LLD. The Pan compiler will by default generate one LLD file per object template found. The name of the LLD file will be the name of the template plus the `.xml` suffix. Object templates cannot be included by any other template.

A *structure template* can only contain assignment and delete statements (and these must act on relative paths). Additionally, it can only include other structure templates. Structure template are used by the `create` function.

[3]These are all the statements except assignment and delete statements.

2.2. STATEMENTS

Here are the three main Pan statements. The others are explained in other (more advanced) sections of this document. The best source of examples is the Appendix part of this document where all the different statements are used in realistic situations.

All the statements (as well as the template declarations) end with a semicolon (i.e. '**;**').

include

statement $\mapsto$ **include** *template-name* '**;**'

The include statement is very similar to `cpp`'s `#include` directive. During the processing, the named template is almost physically included and its statements are executed as if they were in the including template. This can be used to represent some kind of inheritance.

A template may be included multiple times but loops are not allowed. As explained above (see section 2.1.), declaration templates are immune to multiple inclusions.

There are some restrictions on who can include what. 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. The other situations are allowed.

assignement

statement $\mapsto$ *path* '**=**' *dml* '**;**'
path $\mapsto$ *string*

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 of the Data Manipulation Language expression *dml*. This result can be a single property (only one leaf of the tree is changed) or a resource holding any number of elements (a real subtree is changed).

The path is represented by a string literal, see section 3.1.4.. We usually use double quoted strings but any string will do.

If there is no subtree with the given path in configuration tree, it is created with all the necessary parent and lateral elements.

```
# assuming the configuration tree is empty

# /x and /x/a are created, /x/a get the value 1
"/x/a" = 1;

# /y, /y/0 and /y/1 are created, /y/1 get the value 2
# /y/0 has the undefined value (undef)
"/y/1" = 2;

# /y is replaced by a new list of three elements
"/y" = list(4, 5, 6);
```

As explained in the Types section, the Pan compiler always makes sure that the elements that get assigned to have compatible types. See section 4.3. for more information.

delete

statement $\mapsto$ **delete** *path* '**;**'

This statement deletes a subtree of configuration identified by its *path*.
When the parent resource is a list, the subsequent children are moved to fill the hole:

```
# list of three elements
"/list" = list(0, 1, 2);
# we delete the second one
delete "/list/1";
# /list now contains list(0, 2) and not list(0, undef, 2)
```

It is not an error if there is no subtree with the given path in configuration tree. In this case, the statement has no effect.
It is (of course) forbidden to delete the root resource.

Other

In addition to these main statements, there are also others used for: type manipulation (see section 4.), validation (see section 5.1.), user function definition (see section 5.2.) and global variable definition (see section 5.3.).

3. DATA MANIPULATION LANGUAGE

The Data Manipulation Language (*dml* in the syntax definitions) is used to represent subtrees of configuration information, from simple properties to complex resources.
It looks like C but is much simpler and, like in Lisp, every construct returns a value.

3.1. LITERALS

Literal values are the building blocks that can be used to construct more complex expressions. Formally:

$$\boxed{\textit{dml} \mapsto \textbf{true} \mid \textbf{false} \mid \textit{long} \mid \textit{double} \mid \textit{string} \mid \textbf{undef}}$$

Boolean Literals

The boolean literals are very simple, they are only **true** and **false**, with obvious meanings.

Long Literals

The integer number literals are made of digits or of ‘**0x**’ followed by hexadecimal digits, for instance:

```
0
0755
2002
0x20
0xdeadbeef
```

They are stored in what the C compiler knows as the `long` type so too many digits will cause an overflow.
Numbers starting with a leading zero are considered as being octal so a number like `019` is invalid.

Double Literals

The floating point number literals are made of digits followed by a single dot and more digits, or by ‘**e**’ or ‘**E**’ followed by an optional sign and some digits, or by both. For instance:

```
0.01
3.1416
1e-8
6.02217e23
```

They are stored in what the C compiler knows as the `double` type so they will have the same precision.
The part before the dot cannot be omitted so for instance `.2` is not legal and must be written as `0.2`.

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+$'
```

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

```
"foo"
"it's a sentence"
"C 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"
```

Lastly, *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 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 (`base64_decode` is a builtin function):

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

undef

Finally, 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 converted into LLD and most builtin functions will report a fatal error when processing it. It can therefore be used to mark an element that *must* be overwritten during the processing. For instance:

```
# structure template defining a disk
structure template disk_ibm_dtla_307030;
"type"     = "disk";
"vendor"   = "IBM";
"model"    = "DTLA-307030";
"size"     = 29314;
```

```
"serial"   = undef; # must be changed outside of this template

# object template not overwriting serial, will give an error
object template host1;
"devices/hda"  = create("disk_ibm_dtla_307030");

# object template overwriting serial, will process successfully
object template host2;
"devices/hda"  = create("disk_ibm_dtla_307030");
"devices/hda/serial" = "IDCH128457";

# idem but using create with additional parameters
object template host3;
"devices/hda"  = create("disk_ibm_dtla_307030", "serial", "IDCH128457");
```

3.2. FLOW CONTROL

In Pan, three constructs allow you to control the flow of execution. Like in Lisp, they are in fact normal expressions of the Data Manipulation Language so they always return a value.

Sequencing

To execute a sequence of DML instructions, separate them with semicolons and enclose them between curly braces. More formally:

dml	$\mapsto$	'{' *dml-seq* [';'] '}'
dml-seq	$\mapsto$	*dml* [';' *dml-seq*]

The returned value is the value of the last instruction.

Branching

You can use the **if** construct to execute code conditionally. The syntax is:

dml	$\mapsto$	**if** '(' *dml* ')' *dml*
dml	$\mapsto$	**if** '(' *dml* ')' *dml* **else** *dml*

The returned value is the value of the last instruction of the branch that has been executed or **undef** if no branch was executed.

Looping

Finally, you can use the **while** construct to execute some code repeatedly while an expression is true.

dml	$\mapsto$	**while** '(' *dml* ')' *dml*

The returned value is the value of the last instruction of the branch that has been executed or **undef** if no branch was executed.

Examples

```
# set swap variable to be 256 or 512 if RAM is bigger than 256
swap = if (value("/hardware/memory/size") > 256) 512 else 256;
```

```
# nested ifs
if (name == "foo") {
  # do something for foo
  value = 1;
} else if (name == "bar") {
  # do something for bar
  value = 2;
} else {
  # do something else
  value = 3;
};
```

```
# iterate from 0 to 9
i = 0;
while (i <= 9) {
  # do something
  i = i + 1;
};
```

3.3. OPERATORS

Pan's operators are very close to C's. The main difference is that most of them also work on strings, replacing functions such as `strcmp` or `strcat`.

The overall syntax for Pan operators (and grouping) is:

$$\begin{aligned} &\textit{dml} \mapsto \textit{unary-op}\ \textit{dml} \\ &\textit{dml} \mapsto \textit{dml}\ \textit{binary-op}\ \textit{dml} \\ &\textit{dml} \mapsto \text{‘(’}\ \textit{dml}\ \text{‘)’} \end{aligned}$$

The meanings and precedences are identical to C and won't be repeated here.

In the following sections, a *number* means either a long or a double.

Arithmetic Operators

The following operators work on numbers and produce a number:

$$\begin{aligned} &\textit{unary-op} \mapsto \text{‘}-\text{’} \\ &\textit{binary-op} \mapsto \text{‘}+\text{’} \mid \text{‘}-\text{’} \mid \text{‘}*\text{’} \mid \text{‘}/\text{’} \mid \text{‘}\%\text{’} \end{aligned}$$

Note: ‘%’ only works on longs.

String Operator

The ‘+’ operator can also be used on strings and will produce a string which is the concatenation of the two operands.

Comparison Operators

The following operators work on numbers and strings and produce a boolean:

$$\textit{binary-op} \mapsto \text{`<'} \mid \text{`<='} \mid \text{`>'} \mid \text{`>='} \mid \text{`=='} \mid \text{`!='}$$

Note: both operands must be of the same type (i.e. two numbers or two strings), you can't compare a number with a string.

Boolean Operators

The following operators work on booleans and produce a boolean:

$$\textit{unary-op} \mapsto \text{`!'}$$
$$\textit{binary-op} \mapsto \text{`\&\&'} \mid \text{`||'}$$

Note: like in C, '&&' and '||' do "short circuits" and only evaluate their second operand if the result of the first operand is not enough to know the final result.

Example

```
while (i < length(names) && !found) {
  if (names[i] == "foo")
    found = true
  else
    i = i + 1;
};
```

See also Appendix A and Appendix B for more examples.

3.4. FUNCTIONS

Syntax

The overall syntax to call functions (be they builtin or user defined) in Pan is:

$$\textit{dml} \mapsto \textit{identifier}\ \text{`('}\ \text{`)'}$$
$$\textit{dml} \mapsto \textit{identifier}\ \text{`('}\ \textit{argument-seq}\ [\ \text{`,'}\ ]\ \text{`)'}$$
$$\textit{argument-seq} \mapsto \textit{dml}\ [\ \text{`,'}\ \textit{argument-seq}\ ]$$

User defined functions are described in section 5.2..

Builtin Functions

All the builtin functions of the language are described below with the types of their arguments and their return type (when meaningful). When given invalid arguments, they all trigger a fatal compiler error.

base64_decode(*arg:string* **)** *: string*

This function decodes the given string that must be Base64 (defined in RFC 2045) encoded.

```
"/test" = "[" + base64_decode("aGVsbG8gd29ybGQ=") + "]";
# will be the string "[hello world]"
```

---

clone(*arg:element* **)** *: element*

This function returns a clone (copy) of the given argument.

---

create(*name:string, . . .* **)** *: nlist*

This function returns the named list which is the result of the execution of the structure template identified by the given name; the optional extra parameters must be pairs of key and value and will add or modify the result accordingly (a bit like with `nlist`).

```
# description of a CD mount entry (but the device is unknown)
structure template mount_cdrom;
"device"  = undef;
"path"    = "/mnt/cdrom";
"type"    = "iso9660";
"options" = list("noauto", "owner", "ro");

# our first mount entry is the CD coming from hdc
"/system/mounts/0" = create("mount_cdrom", "device", "hdc");

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

---

debug(*message:string* **)**

This function prints the given message on `stdout` when debugging is enabled with the `DEBUG_USER` flag.

```
# print the value of x if it is positive
if (x > 0)
  debug("x is positive: " + to_string(x));
```

---

delete(*arg:variable* **)**

This function deletes the element identified by the ”variable expression” (i.e. variable name with optional subscript such as *foo* or *foo[123]* or *foo[123][”abc”]. . .*).

```
# the following will put the list ("a", "c") at path "/x"
"/x" = {
  x = list("a", "b", "c");
  delete(x[1]);
  return(x);
};
```

---

error(*message:string* **)**

This function prints the given message on `stderr` and aborts the compilation.

```
# user function requiring one long argument
define function foo = {
  if (argc != 1)
    error("foo(): wrong number of arguments: " + to_string(argc));
  if (!is_long(argv[0]))
    error("foo(): argument is not a long");
  # normal processing...
};
```

---

escape(*arg:string* **)** *: string*

This function escape non alphanumeric characters in the given string so that it can be used inside paths, for instance as an named list key.

```
"/test" = escape("1+1");
# will be the string "1_2b1"
```

---

exists(*path:string* **)** *: boolean*

This function checks if the *path* corresponds to an existing element.

---

exists(*arg:variable* **)** *: boolean*

This function checks if the ”variable expression” (see `delete` function) corresponds to an existing element.

---

first(*arg:resource, key:identifier, value:identifier* **)** *: boolean*

This function resets the iterator associated with *arg* so that it points to its first child element; if there is one (i.e. if the resource is not empty), sets the variable *key* to the ”key” of this element (i.e. a long if *arg* is a list or a string if it is a nlist) and the variable *value* to the child element itself and return `true`; if *key* or *value* is `undef`, don’t assign it; if there is no such child element, simply return `false`.

```
# 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);
};
# sum will be 15

# 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")
```

---

index (*sub:string, arg:string* **)** *: long*
index (*sub:string, arg:string, start:long* **)** *: long*

This function 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 only from that position.

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

---

index (*sub:property, arg:list* **)** *: long*
index (*sub:property, arg:list, start:long* **)** *: long*

This function 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 only 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
"/l1" = index("foo", list("Foo", "FOO", "foo", "bar"));
# will be 2

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

---

index (*sub:property, arg:nlist* **)** *: string*
index (*sub:property, arg:nlist, start:long* **)** *: string*

This function 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);

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

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

---

index (*sub:nlist, arg:list* **)** : *long*
index (*sub:nlist, arg:list, start:long* **)** : *long*

This function searches for the given named list inside the given list of named lists 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 only 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
"/l1" = index(nlist("key", "foo"), list(nlist("key", "bar", "val", 101),
                                        nlist("key", "foo", "val", 102)));
# will be 1 (i.e. the second nlist)
```

---

index (*sub:nlist, arg:nlist* **)** : *string*
index (*sub:nlist, arg:nlist, start:long* **)** : *string*

This function searches for the given named list inside the given named list of named lists 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 or if their common children don't have the same type.

---

is_boolean(*arg:element* **)** : *boolean*

This function checks if the given argument is a boolean.

---

is_defined(*arg:element* **)** : *boolean*

This function checks if the given argument is defined (i.e. is anything but `undef`).

---

is_double(*arg:element* **)** : *boolean*

This function checks if the given argument is a double.

---

is_list(*arg:element* **)** : *boolean*

This function checks if the given argument is a list.

---

is_long(*arg:element* **)** : *boolean*

This function checks if the given argument is a long.

---

is_nlist(*arg:element* **)** : *boolean*

This function checks if the given argument is a nlist.

---

is_property(*arg:element* **)** : *boolean*

This function checks if the given argument is a property.

---

is_resource(*arg:element* **)** : *boolean*

This function checks if the given argument is a resource.

---

is_string(*arg:element* **)** : *boolean*

This function checks if the given argument is a string.

---

key(*arg:nlist, index:long* **)** : *string*

This function returns the name of the child identified by its index, this can be used to iterate through all the children of a named list.

```
"/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 "red green blue"
```

---

length(*arg:resource* **)** *: long*

This function returns the number of children of the given resource.

---

length(*arg:string* **)** *: long*

This function returns the length of the given string.

---

list(. . . **)** *: list*

This function returns a list made of its arguments.

```
# the list of mount entries is empty
"/system/mounts = list();
# we define two DNS servers
"/system/dns/servers" = list("137.138.16.5", "137.138.17.5");
# this machine has only one CPU
"/hardware/cpus" = list(create("cpu_intel_p3_850"));
```

---

match(*arg:string, regexp:string* **)** *: boolean*

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

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

---

matches(*arg:string, regexp:string* **)** *: list*

This function matches the given string against the regular expression and returns the list of captured substrings, the first one being the complete matched string (i.e. $& in Perl).

```
# IPv4 address in dotted number notation
define 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);
};
```

---

merge(. . . **)** *: resource*

This function returns the resource which is the merge of its arguments which must be of the same type: either all lists or all named lists; if several named lists have children of the same name, an error occurs.

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

---

next(*arg:resource, key:identifier, value:identifier* **)** *: boolean*

This function increments the iterator associated with *arg* so that it points to the next child element and then behaves like `first`.

---

nlist(. . . **)** *: nlist*

This function returns a named list made of its arguments which must be pairs of key (i.e. child's name, a string) and value (i.e. child's value, an element).

```
# hda1 is our root filesystem
"/system/mounts/0" = nlist("type", "ext2", "path", "/", "device", "hda1");
# hda2 is our /var filesystem
"/system/mounts/1" = nlist(
  "type",   "ext2",
  "path",   "/var",
  "device", "hda2",
);
```

---

return(*arg:element* **)** *: element*

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.

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

---

splice (*arg:list, start:long, length:long* **)** *: list*
splice (*arg:list, start:long, length:long, new:list* **)** *: list*

This function deletes the children of the given list identified by *start* and *length* (see `substr`'s documentation for details) and, if a fourth argument is given, replaces them with the contents of *new*.

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

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

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

---

splice (*arg:string, start:long, length:long* **)** *: string*
splice (*arg:string, start:long, length:long, new:string* **)** *: string*

This function deletes the substring identified by *start* and *length* (see `substr`'s documentation for details) and, if a fourth argument is given, replaces it with *new*.

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

---

substr (*arg:string, start:long* **)** *: string*
substr (*arg:string, start:long, length:long* **)** *: string*

This function returns the part of the given string characterised by its start position (starting from 0) and its length (if 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
```

---

to_boolean(*arg:property* **)** *: boolean*

This function converts the given property into a boolean; 0 and the empty strings are considered as false, everything else as true.

---

to_double(*arg:property* **)** *: double*

This function converts the given property into a double; it is an error if the given string does not represent a double.

---

to_long(*arg:property* **)** *: long*

This function converts the given property into a long; it is an error if the given string does not represent a long; if the argument is a double, it will be rounded to the nearest long.

---

to_string(*arg:property* **)** *: string*

This function converts the given property into a string.

---

unescape(*arg:string* **)** *: string*

This function replace escaped characters in the given string to get back the original string, this is the inverse of the `escape` function.

---

value(*path:string* **)** *: element*

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

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

# we add one DNS server to the current list
# (we need to use list() because merge() requires lists)
"/system/dns/servers" = merge(value("/system/dns/servers"),
  list("137.138.16.5"));

# the RAM of this machine is the same as the machine foo
"/hardware/memory/size" = value("//foo/hardware/memory/size");
```

3.5. 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.

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
};
```

The formal syntax for variable manipulation is:

dml	$\mapsto$	*variable*
dml	$\mapsto$	*variable* '=' *dml*
variable	$\mapsto$	*identifier*
variable	$\mapsto$	*variable* '[' *dml* ']'

A variable holding a resource will in fact hold some kind of pointer to it so variable assignments will simply copy the pointer and not the data. You can use the `clone` function for this purpose. This is illustrated in:

```
"/test" = {
  x = list("a", "b", "c");
  y = x;                      # y point to the same resource as x
  z = clone(x);               # z is a clone/copy of x
  delete(x[1]);               # we modify x by removing its second child
  return(list(length(y),      # this will be 2 as y points to x
              length(z)));    # this will be 3 as z is independent from x
};
```

Variables get automatically destroyed after the execution of the DML expression. Use global variables (see section 5.3.) if you need persistence.

More examples can be found in the Appendix part of this document.

4. TYPES

4.1. INTRODUCTION

Types are very important in Pan. Although no explicit type declaration is required, the compiler permanently knows the type of all the elements it manipulates and makes sure that elements of different types are not mixed by mistake.

In addition, through type declarations, one can impose arbitrary constraints on any part of the configuration tree. When applied to the root of the configuration tree, this can be used to define a ”global data schema”. The compiler, at validaion time, will make sure that the data indeed respects this global schema.

There are two statements concerning types: one to define (name) a new type and one to attach a type to a path. The syntax is:

statement ↦ **define type** *identifier* ‘=’ *type-spec* ‘;’
statement ↦ **type** *path* ‘=’ *type-spec* ‘;’

Types must be known before they are used and cannot be redefined. Several types can be attached to the same path, the compiler will make sure that the data is compatible with all the attached types. The syntax of the type specification (*type-spec*) is explained in the next sections, after the descriptions of the data and builtin types.

4.2. TYPES’ HIERARCHY

Here are the different types in Pan:

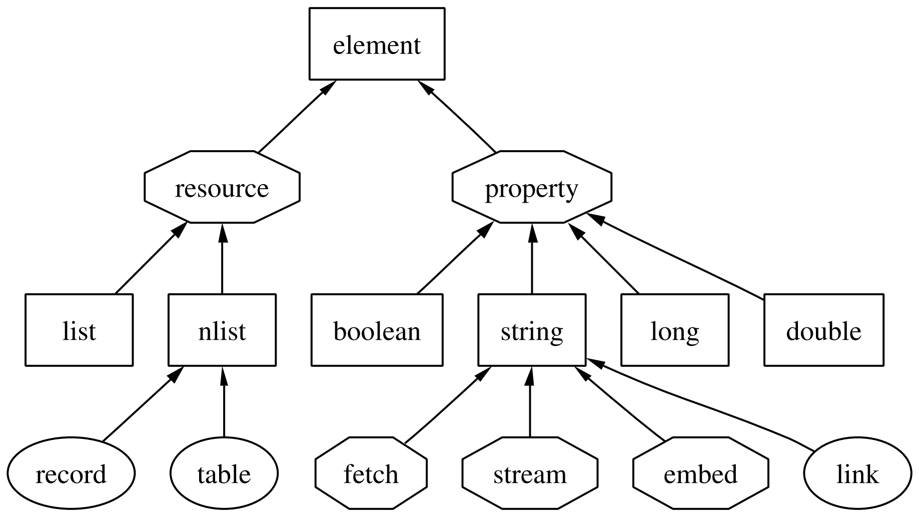

Figure 1: Hierarchy of Pan Types

The arrows denote an ”is a” relation. For instance, a resource is an element and a string is a property.

The types in boxes are the data types, the ones in octagons are builtin while the others are user defined. All are explained below.

4.3. DATA TYPES

The elements that the Pan compiler manipulates all have an intrinsic type which comes from the type of their associated data. For instance, the following line:

```
"/hardware/cpus/0/speed" = 803.5;
```

is enough to deduce that the element at path *"/hardware"* is a nlist, the one at *"/hardware/cpus"* is a list and the one at *"/hardware/cpus/0/speed"* is a double,

There are seven data types which are element, list, nlist (a.k.a. named list), boolean, string, long and double. The *element* data type is associated with the undefined data. For instance, with:

```
"/foo/1" = "test";
"/bar" = undef;
```

both *"/foo/0"* and *"/bar"* have an undefined value and therefore have the *element* data type.

The Pan compiler always makes sure that the elements that get assigned to have the same type (or are undefined). A fatal error occurs otherwise. For instance:

```
              # assuming that /x has never been used before
"/x" = 1;     # /x is now a long
"/x" = "1";   # fatal error: can't assign a string to a long
```

and

```
              # assuming that /x has never been used before
"/x/0" = 1;   # /x is now a list
"/x/a" = 2;   # fatal error: can't access a list by name
```

but

```
              # assuming that /x has never been used before
"/x" = 1;     # /x is now a long
"/x" = undef; # ok as the element type is compatible with everything
"/x" = "1";   # ok as the element type is compatible with everything
```

Assignments of variables are treated exactly the same way:

```
"/test" = {
  x = 1;      # x is now a long
  x = "1";    # fatal error: can't assign a string to a long
};
```

The root resource is a bit special as it is forced to always be a nlist, assigning '**undef**' is forbidden:

```
# forbidden as / would be a list
"/" = list(1, 2, 3);
# forbidden as / would be an (undefined) element
"/" = undef;
# allowed
"/" = create("some_struct_template");
```

If you need to cast an element to another type, you can use the builtin **to**_*type* functions, for instance:

```
"/x" = 1;
"/y" = "/x is " + to_string(value("/x"));
```

The data types are the only ones permanently checked by the compiler, all the others (described in the following sections) are checked only at validation time.

4.4. BUILTIN TYPES

Pan has twelve builtin types that can be used directly in the type statements. They are the seven data types plus:

- *property*: any property
- *resource*: any resource
- *fetch*: a string which is a valid URI
- *stream*: a string which is a valid URI
- *embed*: a string

The last three types have a well defined meaning (see [3]). When associated with an element, they will appear in the generated XML. For instance:

```
type "/x" = fetch;
"/x" = "http://cern.ch/data";
"/y" = "http://cern.ch/data";
```

will generate:

```
<nlist name="profile">
  <string name="x" type="fetch">http://cern.ch/data</string>
  <string name="y">http://cern.ch/data</string>
</nlist>
```

4.5. ALIAS TYPE

There are five ways to define new types based on *existing* ones and therefore five forms of *type-spec*.

The first one is used to give an alias to an existing type, optionally adding some constraints. The syntax is:

type-spec	$\mapsto$	*identifier* [‘**(**’ *long* ‘**)**’] [**with** *dml*]
type-spec	$\mapsto$	*identifier* ‘**(**’ [*long*] ‘**..**’ [*long*] ‘**)**’ [**with** *dml*]

where *identifier* must be the name of an existing type and *dml* some validation code. The *long* bounds (when specified) limit the possible values by imposing minimum and/or maximum values that will be checked directly against the value (for long or double properties), the length (for string properties) or the number of children (for resources). It is illegal to do bound checking on boolean properties. For instance:

```
# long which must be greater or equal to zero
define type ulong1 = long with self >= 0;

# exactly the same thing but defined differently
define type ulong2 = long(0..);

# unsigned short integer (for port numbers)
define type port = long(0..65535);

# string with less than 255 characters
define type short_string = string(..255);

# even long between -16 and +16
define type small_even = long(-16..16) with self % 2 == 0;
```

4.6. LINK TYPE

The second kind of type specification is used to represent links, which are strings containing the path of an *existing* element (at the time of the validation). The link type is defined elsewhere[3]. The syntax is:

type-spec $\mapsto$ *type-spec* '$*$' [**with** *dml*]

The validation code will be run against the string and the linked element must be of the *type-spec* type. For instance:

```
define type mylink = long(0..)* with match(self, 'r$');
type "/foo" = mylink;
"/foo" = "/bar";
"/bar" = 1;
```

will generate:

```
<nlist name="profile">
  <string name="foo" type="link">/bar</string>
  <long name="bar">1</long>
</nlist>
```

The validation in the previous example ensures that the element `"/foo"` is a string that ends with `"r"` and that is also the path of an unsigned long.

4.7. LIST TYPE

You can also define types that represent homogeneous lists. The syntax is:

type-spec $\mapsto$ *type-spec* '**[**' [*long*] '**]**' [**with** *dml*]
type-spec $\mapsto$ *type-spec* '**[**' [*long*] '**..**' [*long*] '**]**' [**with** *dml*]

Elements of this type must be lists of elements of type *type-spec*. The validation code will be run against the list but the min/max constraints apply on the number of elements in the list. For instance:

```
# list of one to four cpu (cpu is a type that must be previously defined)
define type cpus = cpu[1..4];

# list of strings, the first one matching a given regexp
define type spec = string[1..] with match(self[0], '^(start|stop)$');
```

4.8. TABLE TYPE

Similarly to homegeneous lists, you can define homegeneous tables with:

type-spec ↦ *type-spec* '{' [*long*] '}' [**with** *dml*]
type-spec ↦ *type-spec* '{' [*long*] '..' [*long*] '}' [**with** *dml*]

4.9. RECORD TYPE

The last kind of type definition is useful to represent records which are simply nlists with children of known names. The syntax is:

type-spec ↦ '{' *field-spec-seq* '}' [**with** *dml*]
field-spec-seq ↦ *field-spec* [*field-spec-seq*]
field-spec ↦ *string* (':' | '?') *type-spec* [**with** *dml*]
field-spec ↦ **include** *identifier*

Elements of this type must be nlists containing *only* the listed fields, i.e. children with the right name (*string*). The ones marked with '**:**' are mandatory, the ones with '**?**' optional. The children must be of the specified type. The include field specification can be used to extend a record type (i.e. adding more fields). For instance:

```
# record type representing a CPU
define type cpu = {
  "vendor" : string     # name of the vendor
  "model"  : string     # model as known by the vendor
  "speed"  : double     # clock speed in MHz
  "fpu"    ? boolean    # true if it has a FPU
};

# same as cpu but with benchmarking information
define type detailed_cpu = {
  include cpu
  "specint95" : double
  "specfp95"  : double
};
```

4.10. EXAMPLES OF GLOBAL SCHEMA

A very simplified example of global schema can be found in Appendix B. Some compliant XML can be seen in Appendix F.

Some much more complete examples can be found in the WP4 global schema available at `http://edms.cern.ch/document/352656`.

4.11. EMBEDDED DOCUMENTATION

The type specifications described above can also contain some documentation. An optional **description** or **descro** keyword can be used to attach some string to the type definitions. This information is currently ignored by the compiler but it will probably be used in the future to generate some documentation for the global data schema. Here is an example:

```
# record type representing a CPU with embedded documentation
define type cpu = {
  "vendor" : string     description "name of the vendor"
  "model"  : string     description "model as known by the vendor"
  "speed"  : double     description "clock speed in MHz"
  "fpu"    ? boolean    description "true if it has a FPU"
};
```

This feature is also used in the example in Appendix G.

5. ADVANCED FEATURES

5.1. VALIDATION

Arbitrary validation code can be attached to any part of the configuration tree. The syntax is:

statement ↦ **valid** *path* '=' *dml* ';'

At the end of the processing and before generating the LLD, the compiler will run all the attached validation codes. If any of these does not return the true property, the validation fails and the compiler aborts with an error message.

The given *path* must correspond to an existing element (at validation time), otherwise an error is reported. This element can be used in the validation code via the `self` variable:

```
# make sure that the machine has at least 256MB of RAM per CPU
valid "/hardware/memory/size" =
  self >= 256 * length(value("/hardware/cpus"));
```

The validation code is not allowed to modify the configuration tree. The tree is effectively read-only and any attempt to modify it will trigger a fatal error.

The `error` builtin function can also be used to give more user friendly error messages. For instance:

```
object template test;
valid "/test" =
  self > 10 || error("too small: " + to_string(self));
"/test" = 7;
# will give: validation error: user error: too small: 7
```

When combined with type definitions and user defined functions, the validation can easily enforce arbitrary constraints on the generated LLD. The Appendix G shows an example of this where we make sure that NFS servers indeed export the directories that NFS clients mount.

5.2. USER DEFINED FUNCTIONS

In addition to the builtin functions, Pan can also handle user defined functions. The syntax to define a new function is:

statement ↦ **define function** *identifier* '=' *dml* ';'

All functions (builtin and user defined) reside in one global namespace and cannot be redefined.

At invocation time, the given *dml* will be executed and its result will be the result of the function. One can also use the `return` function to return immediately from the *dml*.

Recursion is allowed but is limited to some arbitrary low limit. Pan is not a full blown programming language and complicated functions are often a sign that something is wrong. Maybe another data schema would simplify a lot the data handling.

The function parameters can be accessed via the `argc` variable (holding the number of parameters passed) and `argv` (holding the parameters themselves). Up to the function code to check that the parameters are indeed what is expected. For instance:

```
# insert a string after another one in a list of strings
# (or at the end if not found)
define function insert_after = {
  if (argc != 3 || !is_string(argv[0]) || !is_string(argv[1]) ||
      !is_list(argv[2]))
    error("usage: insert_after(string, string, list)");
  idx = index(argv[1], argv[2]);
  if (idx < 0) {
    # not found, we insert at the end
    splice(argv[2], length(argv[2]), 0, list(argv[0]));
  } else {
    # found, we insert just after
    splice(argv[2], idx+1, 0, list(argv[0]));
  };
  return(argv[2]);
};
```

The function parameters can be modified from within the function code. The following function increments its first parameter by the value of the second one:

```
define function increment = {
  argv[0] = argv[0] + argv[1];
};

"/test" = {
  x = 1;
  increment(x, 2);
  return(x); # this will be 3
};
```

More examples can be found in the Appendix A.

5.3. GLOBAL VARIABLES

In addition to local variables, Pan can also handle global variables:

statement $\mapsto$ **define variable** *identifier* '=' *dml* ';'

This statement defines a global variable named *identifier* and gives it the value *dml*. Variables can be redefined, this effectively only change their value.

Global variables can be read from any DML block but they cannot be modified from the DML. The following lines will generate an error:

```
define variable foo = 1;

"/test" = {
  foo = 2;     # error
  return(3);
};
```

Pan provides two global variables. The first one is named `object` and contains the name of the object template being processed. The second one is named `self` and contains the element being processed, i.e. being assigned to (for assignments) or being validated (for validation code).

Global variables are very useful when combined with declaration templates as they can represent some values that can be changed without giving write access to the rest of the configuration tree. For instance:

```
# a template that the machine user can modify
declaration template host17_user;
define variable resolution = "1024x768";

# a template that the machine sysadmin can modify
object template host17;
define variable resolution = "1280x1024";
include host17_user;
"/system/x/resolution" = resolution;
```

If the system administrator of `host17` can indeed modify the template `host17` while the machine user can only modify `host17_user`, this guarantees that the user can effectively only change the X server resolution from its default value. Any attempt to directly modify the configuration tree in the declaration template is forbidden by the language.

6. FOR MORE INFORMATION

6.1. GLOSSARY

- a *configuration information* is any piece of information that is needed in order to statically configure a machine. It does not include dynamic information that changes while the machine is normally running (e.g. the contents of a database hosted on the machine) and information generated by the machine itself such as system load or the fact that the machine is being reconfigured.
- the *configuration database* (CDB) is the system holding configuration information for a given set of machines (e.g. for a computer centre or for a whole site). It provides different views on the stored information, optimised for the different access patterns of the programs requesting configuration information.
- the *high-level description* (HLD) is the view optimised for high-level operations such as configuration management of a large number of machines: it's read-write and supports abstraction.
- the *node view* or low-level description (LLD) is the view optimised for normal machine configuration operations: it's read-only but scalable and contains only the configuration information relevant to the machine requesting it.
- a *configuration element* is a piece of configuration information. It may be a configuration property or a configuration resource. Configuration elements form a tree structure.
- a *property* is a configuration element that holds a simple value. It is a leaf element in the tree of configuration information formed by the configuration elements.
- a *resource* is a configuration element that has other elements as children. It is an interior element in the tree of configuration information formed by the configuration elements.
- a *list* is a resource that contains unnamed children elements.
- a *nlist* is a resource that contains named children elements. Two children cannot have the same name.
- *path* is a identifier of a configuration element in the tree of configuration information.

6.2. ACRONYMS

- *CCM*: Configuration Cache Manager
- *CDB*: Configuration DataBase
- *HLD*: High-Level Description
- *LLD*: Low-Level Description
- *XML*: eXtensible Markup Language
- *UNC*: Universal Naming Convention

6.3. REFERENCES

[1] Lionel Cons and Piotr Poznański. Configuration Database Global Design, 2002. http://cern.ch/hep-proj-grid-fabric-config.

[2] European Union DataGrid Project (EDG). http://www.eu-datagrid.org.

[3] Michael George. Node Profile Specification, 2002. http://cern.ch/hep-proj-grid-fabric-config.

[4] EDG Fabric Management Work Package (WP4). http://cern.ch/hep-proj-grid-fabric.

[5] EDG WP4 Configuration Task (WP4C). http://cern.ch/hep-proj-grid-fabric-config.

A SAMPLE FUNCTIONS

```
###############################################################################
#
# Useful functions.
#
# $Id: functions.tpl,v 1.7 2002/08/12 07:55:04 cons Exp $
#
###############################################################################

declaration template functions;

#
# insert_after(string, string, list): insert the first string after the second
# one (if found) or at the end (otherwise); the last argument is modified but
# also returned as the result of the function
#

define function insert_after = {
  if (argc != 3 ||
      !is_string(argv[0]) || !is_string(argv[1]) || !is_list(argv[2]))
    error("usage: insert_after(string, string, list)");
  idx = index(argv[1], argv[2]);
  if (idx < 0) {
    # not found, we insert at the end
    splice(argv[2], length(argv[2]), 0, list(argv[0]));
  } else {
    # found, we insert just after
    splice(argv[2], idx+1, 0, list(argv[0]));
  };
  return(argv[2]);
};

#
# given a disk name, return a table of three primary partitions for swap, root
# and /var with a very simple space allocation algorithm
#

define function simple_partitions = {
  if (argc != 1 || !is_string(argv[0]))
    error("usage: simple_partitions(string)");
  disk = argv[0];
  disk_size = value("/hardware/devices/" + disk + "/size");
  # swap is twice the size of the physical memory
  swap = nlist(
    "disk", disk,
    "type", "primary",
    "size", 2 * value("/hardware/memory/size"),
    "id",   82, # Linux swap
  );
  # var is 256MB for disks larger than 2GB, 128MB otherwise
  var = nlist(
    "disk", disk,
    "type", "primary",
    "size", if (disk_size > 2048) 256 else 128,
    "id",   83, # Linux
  );
  # root is the rest
  root = nlist(
    "disk", disk,
    "type", "primary",
    "size", disk_size - swap["size"] - var["size"],
    "id",   83, # Linux
  );
  # order of partitions is swap, root and var
  return(nlist(
    disk+"1", swap,
    disk+"2", root,
    disk+"3", var,
  ));
};
```

B SAMPLE TYPES

```
##############################################################################
#
# Useful (but simplified) types.
#
# $Id: types.tpl,v 1.4 2002/07/23 09:35:35 cons Exp $
#
##############################################################################

declaration template types;

##############################################################################
#
# simple types
#

# unsigned long
#(old style) define type ulong = long with self >= 0;
define type ulong = long(0..);

# unsigned double
define type udouble = double(0..);

# IPv4 address in dotted number notation
define 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);
};

##############################################################################
#
# hardware types
#

# memory record
define type memory_t = {
  "size" : ulong
};

# CPU record
define type cpu_t = {
  "vendor" : string
  "model"  : string
  "speed"  : udouble
};

# device record (describing some hardware devices such as disks)
define type device_t = {
  "type"     : string with match(self, '^(disk|cd|net)$')
  "vendor"   : string
  "model"    : string
  "size"     ? ulong
  "driver"   ? string
  "address"  ? string
};

# hardware record (describing some complete hardware information)
define type hardware_t = {
  "vendor"  : string
  "model"   : string
  "serial"  : string
  "memory"  : memory_t
  "cpus"    : cpu_t[1..]   # list of at least one CPU
  "devices" : device_t{}   # table of devices, indexed by names such as hda
};

##############################################################################
#
# system types
#

# mount record (describing what will end up in /etc/fstab)
define type mount_t = {
  "device"  ? string
  "path"    : string
  "type"    : string
  "name"    ? string
  "options" ? string[]
};

# partition record (describing how to partition the disks)
define type partition_t = {
  "disk" : string with value("/hardware/devices/"+self+"/type") == "disk"
  "type" : string
  "size" : ulong
  "id"   : ulong
};

# system record (describing some of the system configuration)
define type system_t = {
  "mounts"     : mount_t[1..]   # list of at least one mount
  "partitions" ? partition_t{}  # table of partitions, indexed by e.g. hda1
};

##############################################################################
#
# root type
#

# type of the root of the configuration information
define type root_t = {
  "hardware" : hardware_t    # hardware subtree
  "system"   : system_t      # system subtree
};

# declare that root is indeed of the root type
type "/" = root_t;
```

C SAMPLE STRUCTURE TEMPLATES

```
###############################################################################
#
# Sample hardware data.
#
# $Id: hardware.tpl,v 1.5 2002/07/23 09:36:34 cons Exp $
#
###############################################################################

###############################################################################
#
# cpus
#

structure template cpu_intel_p3_800;
"vendor" = "Intel";
"model"  = "Pentium III (Coppermine)";
"speed"  = 796.550; # MHz

structure template cpu_intel_p3_850;
"vendor" = "Intel";
"model"  = "Pentium III (Coppermine)";
"speed"  = 853.220; # MHz

###############################################################################
#
# disks
#

structure template disk_quantum_fireballp_as20_5;
"type"     = "disk";
"vendor"   = "QUANTUM";
"model"    = "FIREBALLP AS20.5";
"size"     = 19596; # MB

structure template disk_ibm_dtla_307030;
"type"     = "disk";
"vendor"   = "IBM";
"model"    = "DTLA-307030";
"size"     = 29314; # MB

###############################################################################
#
# cdroms
#

structure template cdrom_lg_crd_8521b;
"type"     = "cd";
"vendor"   = "LG";
"model"    = "CRD-8521B";

###############################################################################
#
# network cards
#

structure template network_3com_3c905b;
"type"     = "net";
"vendor"   = "3Com";
"model"    = "3c905B-Combo [Deluxe Etherlink XL 10/100]";
```

```
"driver"   = "3c59x";

structure template network_intel_82557;
"type"     = "net";
"vendor"   = "Intel";
"model"    = "82557 [Ethernet Pro 100]";
"driver"   = "eepro100";

###############################################################################
#
# computers
#

structure template pc_elonex_850_256;
"vendor"       = "Elonex";
"model"        = "850/256";
"cpus"         = list(create("cpu_intel_p3_850"));
"memory/size"  = 256; # MB
"devices/hda"  = create("disk_quantum_fireballp_as20_5");
"devices/hdc"  = create("cdrom_lg_crd_8521b");
"devices/eth0" = create("network_3com_3c905b");

structure template pc_elonex_800x2_512;
"vendor"       = "Elonex";
"model"        = "800x2/512";
"cpus"         = list(create("cpu_intel_p3_800"), create("cpu_intel_p3_800"));
"memory/size"  = 512; # MB
"devices/hda"  = create("disk_ibm_dtla_307030");
"devices/eth0" = create("network_intel_82557");
```

D SAMPLE TEMPLATES

```
################################################################################
#
# Sample system data.
#
# $Id: system.tpl,v 1.3 2002/07/26 09:47:03 cons Exp $
#
################################################################################

################################################################################
#
# standard mounts
#

structure template mount_afs;
"path"    = "/afs";
"type"    = "afs";

structure template mount_proc;
"path"    = "/proc";
"type"    = "proc";

structure template mount_devpts;
"path"    = "/dev/pts";
"type"    = "devpts";
"options" = list("gid=5", "mode=620");

structure template mount_floppy;
"device"  = "fd0";
"path"    = "/mnt/floppy";
"type"    = "ext2";
"options" = list("noauto", "owner");

structure template mount_cdrom;
"device"  = undef;
"path"    = "/mnt/cdrom";
"type"    = "iso9660";
"options" = list("noauto", "owner", "ro");

################################################################################
#
# mounting templates
#

# add the standard Linux mount entries
template mounting_linux;
"/system/mounts" = merge(value("/system/mounts"), list(
  create("mount_proc"),
  create("mount_devpts"),
  create("mount_floppy"),
));

# add the AFS mount entry
template mounting_afs;
"/system/mounts" = merge(value("/system/mounts"), list(create("mount_afs")));
```

E SAMPLE OBJECT TEMPLATES

```
##############################################################################
#
# Sample object template.
#
# $Id: sample.tpl,v 1.4 2002/08/12 07:57:49 cons Exp $
#
##############################################################################

object template sample;

# standard includes
include types;
include functions;

# hardware information
"/hardware" = create("pc_elonex_850_256");
"/hardware/serial" = "CH01112041";
"/hardware/devices/eth0/address" = "00:d0:b7:a9:a3:47";

# system information
"/system/partitions" = simple_partitions("hda");
"/system/mounts/0" = nlist("type" , "swap", "path", "swap", "device", "hda1");
"/system/mounts/1" = nlist("type" , "ext2", "path", "/",    "device", "hda2");
"/system/mounts/2" = nlist("type" , "ext2", "path", "/var", "device", "hda3");
include mounting_linux;
include mounting_afs;

# we also add a mount entry for our CD drive...
"/system/mounts" = merge(value("/system/mounts"),
                         list(create("mount_cdrom", "device", "hdc")));
# ...and make sure that hdc indeed contains a CD drive!
valid "/hardware/devices/hdc" = self["type"] == "cd";
```

F SAMPLE XML

```
<?xml version="1.0" encoding="utf-8"?>
<nlist name="profile" type="record">
  <nlist name="hardware" type="record">
    <string name="vendor">Elonex</string>
    <string name="model">850/256</string>
    <list name="cpus">
      <nlist type="record">
        <string name="vendor">Intel</string>
        <string name="model">Pentium III (Coppermine)</string>
        <double name="speed">853.22</double>
      </nlist>
    </list>
    <string name="serial">CH01112041</string>
    <nlist name="memory" type="record">
      <long name="size">256</long>
    </nlist>
    <nlist name="devices" type="table">
      <nlist name="hda" type="record">
        <string name="type">disk</string>
        <long name="size">19596</long>
        <string name="vendor">QUANTUM</string>
        <string name="model">FIREBALLP AS20.5</string>
      </nlist>
      <nlist name="hdc" type="record">
        <string name="type">cd</string>
        <string name="vendor">LG</string>
        <string name="model">CRD-8521B</string>
      </nlist>
      <nlist name="eth0" type="record">
        <string name="type">net</string>
        <string name="vendor">3Com</string>
        <string name="model">3c905B-Combo [Deluxe Etherlink XL 10/100]</string>
        <string name="driver">3c59x</string>
        <string name="address">00:d0:b7:a9:a3:47</string>
      </nlist>
    </nlist>
  </nlist>
  <nlist name="system" type="record">
    <list name="mounts">
      <nlist type="record">
        <string name="type">swap</string>
        <string name="path">swap</string>
        <string name="device">hda1</string>
      </nlist>
      <nlist type="record">
        <string name="type">ext2</string>
        <string name="path">/</string>
        <string name="device">hda2</string>
      </nlist>
      <nlist type="record">
        <string name="type">ext2</string>
        <string name="path">/var</string>
        <string name="device">hda3</string>
      </nlist>
      <nlist type="record">
        <string name="type">proc</string>
        <string name="path">/proc</string>
      </nlist>
      <nlist type="record">
```

```
          <string name="type">devpts</string>
          <string name="path">/dev/pts</string>
          <list name="options">
            <string>gid=5</string>
            <string>mode=620</string>
          </list>
        </nlist>
        <nlist type="record">
          <string name="type">ext2</string>
          <string name="path">/mnt/floppy</string>
          <string name="device">fd0</string>
          <list name="options">
            <string>noauto</string>
            <string>owner</string>
          </list>
        </nlist>
        <nlist type="record">
          <string name="type">afs</string>
          <string name="path">/afs</string>
        </nlist>
        <nlist type="record">
          <string name="type">iso9660</string>
          <string name="path">/mnt/cdrom</string>
          <string name="device">hdc</string>
          <list name="options">
            <string>noauto</string>
            <string>owner</string>
            <string>ro</string>
          </list>
        </nlist>
      </list>
      <nlist name="partitions" type="table">
        <nlist name="hda1" type="record">
          <string name="disk">hda</string>
          <string name="type">primary</string>
          <long name="size">512</long>
          <long name="id">82</long>
        </nlist>
        <nlist name="hda2" type="record">
          <string name="disk">hda</string>
          <string name="type">primary</string>
          <long name="size">18828</long>
          <long name="id">83</long>
        </nlist>
        <nlist name="hda3" type="record">
          <string name="disk">hda</string>
          <string name="type">primary</string>
          <long name="size">256</long>
          <long name="id">83</long>
        </nlist>
      </nlist>
    </nlist>
</nlist>
```

G EXAMPLE OF CROSS OBJECT VALIDATION

```
##############################################################################
#
# Simplified example of cross object validation.
#
# All the NFS clients check that the NFS servers that they use indeed export
# the directories that they mount. This is done transparently by adding some
# validation code to the mount record type. Further checks such as wildcards
# in export list or export/mount options mismatch are left as an exercise for
# the reader ;-)
#
# Here is how to compile the server and two clients (result on stdout):
#   % pan --stdout --output=nfssrv1 xvalidation.tpl             (will succeed)
#   % pan --stdout --output=nfsclt1 xvalidation.tpl             (will succeed)
#   % pan --stdout --output=nfsclt2 xvalidation.tpl             (will fail)
#
# $Id: xvalidation.tpl,v 1.5 2002/07/23 09:24:30 cons Exp $
#
##############################################################################

##############################################################################
#
# types definitions
#

template types;

# export record (roughly what is in /etc/exports)
define type export = {
  "path"    : string   description "path of the exported directory"
  "client"  : string   description "name of client allowed to mount it"
  "options" ? string[] description "list of exporting options like ro"
};

# mount record (roughly what is in /etc/fstab)
define type mount = {
  "device"  : string   description "device as understood by the mount command"
  "path"    : string   description "path of the mount point"
  "type"    : string   description "type of the mounted filesystem"
  "name"    ? string   description "name or label of this mount entry"
  "options" ? string[] description "list of mounting options like ro"
} with valid_mount(self);

# validation of a mount record (only nfs type records are checked)
define function valid_mount = {
  # the mount record is our only argument
  mount = argv[0];
  # we only care about NFS mounts, other types are considered OK
  if (mount["type"] != "nfs")
    return(true);
  # the device field will give us the NFS server and path
  result = matches(mount["device"], '^([\w\.\-]+):(.+)$');
  if (length(result) == 0)
    error("bad nfs device: " + mount["device"]);
  server = result[1];
  path   = result[2];
  # we now look at the server's exports list
  exports = value("//" + server + "/system/exports");
  i = 0;
  len = length(exports);
  while (i < len) {
    # we check if this export record is good for us by checking the client
    # field against object (i.e. the name of the current object template)
    # and the path; we want exact match and ignore the export/mount options
    if (exports[i]["client"] == object && exports[i]["path"] == path)
      return(true);
    i = i + 1;
  };
  # we haven't found any export record matching our needs, we complain:
  error("server " + server + " does not export " + path + " to " + object);
};

###############################################################################
#
# NFS server definition
#

object template nfssrv1;

# type settings
include types;
type "/system/exports" = export[];

# data for this host
"/system/exports" = list(
  nlist(                      # we export /home to hostx
    "path",    "/home",
    "client",  "hostx",
  ),
  nlist(                      # we export /home to nfsclt1, read-only
    "path",    "/home",
    "client",  "nfsclt1",
    "options", list("ro"),
  ),
);

###############################################################################
#
# NFS clients definitions
#

template client;

# type settings
include types;
type "/system/mounts" = mount[];

# data for this host
"/system/mounts" = list(
  nlist(                      # we mount /dev/hda1 as the root filesystem
    "device", "/dev/hda1",
    "path",   "/",
    "type",   "ext2",
  ),
  nlist(                      # we NFS mount /home from the server nfssrv1
    "device", "nfssrv1:/home",
    "path",   "/home",
    "type",   "nfs",
  ),
);
```

```
# first client: known by the server, compilation will succeed
object template nfsclt1;
include client;

# second client: unknown to the server, compilation will fail with:
# *** user error: server nfssrv1 does not export /home to nfsclt2
object template nfsclt2;
include client;
```

H VERSION HISTORY

2.0.2 (April 2, 2003): template names are now less restricted (see section 1.4.).

2.0.1 (November 14, 2002): added more builtin functions (`escape`, `unescape`, `first` and `next`), improved the `index` builtin function to search for nlists, extended the scope of the `self` variable (see section 5.3.).

2.0.0 (October 2, 2002): initial version of this document.