Pan Tutorial

Charles Loomis

Abstract

This tutorial introduces the pan compiler and pan configuration language. Those following the tutorial should understand the purpose of the pan compiler and how the pan configuration language allows a system administrator to describe the configuration of a machine. This tutorial highlights all of the major features of the pan language: simple declarative syntax, type checking, and extensive schema validation.

Table of Contents

1. Purpose of the Pan Compiler

The pan compiler is a critical component of the quattor fabric management toolkit that translates a high-level site configuration written by a system administrator in the pan configuration language to

a machine-readable representation. The pan configuration language allows a system administrator to define simultaneously both a site configuration and a schema for validation. One advantage the pan language has over, for example, XML and XMLSchema is that it is declarative language with a simple, human-friendly syntax. In addition, the typing features of the pan language allow more rigorous validation than XMLSchema.

The name "compiler" is actually a misnomer, as the pan compiler does much more than a simple compilation. The processing progresses through five stages:

compilation	Compile each individual template (file written in the pan configuration language) into a binary format.
execution	The statements in each object template are executed to generate a partial tree of configuration information. The object template usually includes many other templates during the course of execution. The generated tree contains all configuration information directly specified by the system administrator.
insertion of defaults	A pass is made through the tree of configuration information during which default values (if specified) are inserted for missing elements. The tree of configuration information is complete after this stage.
validation	The configuration information is frozen and all of the user-specified validation is run. (This is actually implemented as two separate validation passes to allow circular validation dependencies between machines.) Any invalid values or conditions will cause the processing to abort. If this stage finishes, then the configuration information is complete and validated.
serialization	Once the information is complete and valid, it is serialized to a file. Usually, this file is in an XML format, but other representations are available as well.

The pan compiler runs through these stages for each "object" template. An "object" template is a special template that indicates a configuration tree that should be serialized; usually there is one object template for each physical machine. (Although with the rise of virtualization, it may be one per logical machine.)

2. Invoking the Pan Compiler

In order to experiment with the pan compiler, you must first install it; see the pan compiler manual for instructions for downloading and installing an appropriate version. For this tutorial, the command line interface is the most convenient method for invoking the compiler. However, there are other mechanisms for invoking it directly from java or for integrating it with ant.

Once you have installed the compiler, make sure that it is correctly installed by using the commands:

```
$ panc --version
$ panc --help
```

The first command will return the version of the compiler; the second will give a complete list of all of the available options.

Now create a file named `hello.tpl` that contains the following:

```
object template hello;
'/message' = 'hello';
```

Now compile this profile into the default XML representation and look at the output.

```
$ panc hello.tpl
```

```
$ cat hello.xml
<?xml version="1.0" encoding="UTF-8"?>
<nlist format="pan" name="profile">
<string name="message">hello</string>
</nlist>
```

The output should look similar to what is shown above. As you can see the generated information has a simple structure: a top-level element of type nlist, named "profile" with a single string child, named "message". The value of the "message" is "hello". If the output format is not specified, the default is the "pan" XML style shown above, in which the element names are the pan primitive types and the name attribute gives the associated name of the element in the pan template.

The pan compiler can generate output in three additional formats: xmldb, text, and dot. The xmldb format is a format in which the pan names are used for XML elements and the type attribute gives the element type. (If not specified, the default type is string.) The following shows the output for the xmldb format.

```
$ panc --xml-style=xmldb hello.tpl
$ cat hello.xml
<?xml version="1.0" encoding="UTF-8"?>
<profile format="xmldb">
<message type="string">hello</message>
</profile>
```

This format is convenient if the resulting output will be processed with tools like XSLT or XQuery because the XPath expression to use for a particular element is very close (but not identical) to the pan language path.

For this tutorial, the most convenient representation will the the text format. This provides a clean representation of the configuration tree on a terminal.

```
$ panc --xml-style=text hello.tpl
$ cat hello.txt
+-profile
        $ message : (string) 'hello'
```

Note that the output file is named `hello.txt` and no longer `hello.xml`. It provides the same information as the XML formats, but is easier to understand visually.

The last style is the "dot" format. This format produces an output file that Graphviz [http://www.graphviz.org/] can use to generate a graph of the configuration information.

```
$ panc --xml-style=dot hello.tpl
$ cat hello.dot
digraph "profile" {
bgcolor = beige
node [ color = black, shape = box, fontname=Helvetica ]
edge [ color = black ]
"/profile" [ label = "profile" ]
"/profile/message" [ label = "message\n'hello'" ]
"/profile" -> "/profile/message"
}
```

Although the text is not very enlightening by itself, it can be used by Graphviz to generate a graph of the configuration. Processing the above file with Graphviz produces the image shown in Figure 1, “Graph of configuration produced by hello.tpl.”. The images in the tutorial have been produced with the dot output of the compiler.

Figure 1. Graph of configuration produced by `hello.tpl`.

3. Tutorial Scenario

This tutorial will demonstrate how to define a configuration schema and populate it with values using the pan configuration language. To keep this tutorial from becoming a dry exercise of pan functionality, a simplified but typical configuration scenario will be used. A system administrator has four machines that need to be installed. One of the machines will be an NFS server to provide a shared file system to the other three machines. One of the machines will be the head node of a batch system and the other two will be the batch system clients, or worker nodes.

The corresponding configuration schema will consist of both hardware and software aspects of the configuration. For the hardware aspects, the schema will include information about the location of the machine, RAM, CPUs, disks, and network interface cards (NICs). The software configuration will include parameters for configuring an NFS server, NFS client, batch system head node, and batch system client. The software configuration will also contain information about some related low-level services such as a firewall.

The schema developed in this tutorial borrows ideas from the standard Quattor schema. However for pedagogical reasons, the schema here is simplified to demonstrate the essential pan language features. At the end of the tutorial, readers should understand the core features of the pan configuration language and be able to understand the standard Quattor schema. Additional work with other components of the Quattor system will be necessary to use Quattor to configure a real set of machines.

4. Setting Configuration Values

The pan configuration language is a declarative language that allows configuration parameters arranged in a hierarchical tree to be given values. At its simplest level, pan simply sets key/value pairs where the keys can be arranged hierarchically. Consider initially the hardware information for the machines: location of the machine, RAM, CPUs, disks, and network interface cards (NICs). One possible way of specifying this information in pan is the following:

```
object template nfsserver.example.org;

'/hardware/location/rack' = 'IBM04';
'/hardware/location/slot' = 25;

'/hardware/ram' = 2048;

'/hardware/cpu/model' = 'Intel Xeon';
'/hardware/cpu/speed' = 2.5;
'/hardware/cpu/arch' = 'x86_64';
'/hardware/cpu/cores' = 4;
'/hardware/cpu/number' = 2;

'/hardware/disks/ide/0/capacity' = 64;
'/hardware/disks/ide/0/boot' = true;
'/hardware/disks/ide/0/label' = 'system';
'/hardware/disks/ide/1/capacity' = 1024;
'/hardware/disks/ide/1/boot' = false;

'/hardware/nic/0/mac' = '01:23:45:ab:cd:99';
'/hardware/nic/0/pxeboot' = false;
'/hardware/nic/1/mac' = '01:23:45:ab:cd:00';
'/hardware/nic/1/pxeboot' = true;
```

In this example, each assignment statement sets one value. On the left-hand side is the absolute path, which must be a single- or double-quoted string; the right-hand side is the value to assign to that path. All absolute paths must begin with a slash. Save this to the file `nfsserver.example.org.tpl`.

Invoking the pan compiler on this will cause a configuration tree to be built by executing the assignment statements in order from the beginning to the end. The text representation of the compiler output looks like:

```
$ panc --xml-style=text nfsserver.example.org
$ cat nfsserver.example.org.txt
+-profile
        +-hardware
                +-cpu
                        $ arch : (string) 'x86_64'
                        $ cores : (long) '4'
                        $ model : (string) 'Intel Xeon'
                        $ number : (long) '2'
                        $ speed : (double) '2.5'
                +-disks
                        +-ide
                                +-0
                                        $ boot : (boolean) 'true'
                                        $ capacity : (long) '64'
                                        $ label : (string) 'system'
                                +-1
                                        $ boot : (boolean) 'false'
                                        $ capacity : (long) '1024'
                +-location
                        $ rack : (string) 'IBM04'
                        $ slot : (long) '25'
                +-nic
                        +-0
                                $ mac : (string) '01:23:45:ab:cd:99'
                                $ pxeboot : (boolean) 'false'
                        +-1
                                $ mac : (string) '01:23:45:ab:cd:00'
                                $ pxeboot : (boolean) 'true'
                $ ram : (long) '2048'
```

showing how a machine with an dual-CPU machine with Intel Xeon chips running at 2.5 GHz, 2 GB of RAM, two IDE disks, two NICs, and located in rack IBM04 in slot 25 could be represented. This information could then be used by other parts of the Quattor toolkit (or any other tool for that matter) to do some work based on this information, although the XML format is usually used when transmitting the information to other tools.

Although this example should be fairly intuitive for most readers, there are a few subtleties to point out. The first line of the example indicates that this file represents a "managed object" and should produce an output file. If the `object` modifier is left out, then the file will compile but no output file will be produced. The tutorial will explain the different types of templates later.

Notice that it was not necessary to specify all of the parents of a particular path. For example, there was no assignment statement with the path `/hardware`. When assigning a value to a particular path, pan will automatically create the parents as necessary. In the pan language, the leaves of the tree (terminal values) are called *properties*. All of the assignment statements in this example set properties. The branches of the tree (collections) are generically called *resources*. The term *element* encompasses both pan properties and resources.

4.1. Properties and Primitive Types

The properties can have any of the primitive types that pan supports: long, double, boolean, and string. The syntax for writing literal values is the same as most modern programming languages. In addition to the usual base-10 format for long literals, octal and hexadecimal literals can also be specified by using a leading '0' or '0x' respectively. For example, fifteen can be written as `15`, `017`, or `0xf`. Double

literals can be specified with or without an exponent. One particularity of pan is that all double literals must start with a digit. That is, `.2` is not a valid double literal and must instead be written as `0.2`.

String values may be specified with a single-quoted, double-quoted, or a *heredoc* syntax. All characters within a single-quoted string are taken literally. For example, the string `'no new line\n'` will contain a backslash and character `n` at the end of the string. The only exception is that a doubled single quote represents a single literal quote within a single-quoted value. It is best practice to use single-quoted strings to specify path values. In contrast, the double-quoted string `"with a new line\n"` will contain a new line character at the end. All of the usual c-style escape sequences are supported. For long multi-line strings, the *heredoc* syntax may be used:

```
object template heredoc;

'/longstring' = <<EOF;
This is a
long multiline
string.
EOF
```

The token after the `<<` operator will mark the end of the multi-line string and can be chosen by the user. The string will begin after the next new line in the source template and end before the line containing the end token. The ending token must appear on a separate line by itself; no other characters (including whitespace) may appear on the line with the ending token.

4.2. Resources

There are two types of *resources* supported by pan: list and nlist. A list is an ordered list of elements with the indexing starting at zero. In the above example, there are two lists `/hardware/disks/ide` and `/hardware/nic`. The order of a list is significant and maintained in the serialized representation of the configuration. An nlist (named list) associates a name with an element; these are also known as hashes or associative arrays. One nlist in the above example is `/hardware/cpu`, which has `arch`, `cores`, `model`, `number`, and `speed` as children. Note that the order of an nlist is *not* significant and that the order specified in the template file is *not* preserved in the serialized version of the configuration. Although the algorithm for ordering the children of an nlist in the serialized file is not specified, the pan compiler guarantees a *consistent* ordering of the same children from one compilation to the next.

Within a given path, lists and nlists can be distinguished by the names of their children. Lists always have children whose names are valid long literals. In the following example, /mylist is a list with three children:

```
object template mylist;

'/mylist/0' = 'decimal index';
'/mylist/01' = 'octal index';
'/mylist/0x3' = 'hexadecimal index';
```

The indices can be specified in decimal, octal, or hexadecimal. The names of children in an nlist must begin with a letter or underscore.

4.3. Exercises

1. Pan creates parent resources automatically inferring the type of resource (list or nlist) from the path itself. What happens if two assignment statements implicitly define different types for a resource?

2. What happens if a list is defined with a "hole" in it? Try defining and compiling a template in which a list only has the third element (index = 2) defined.

3. What happens if you redefine the same path later in the same template? Which value appears in the serialized file?

4. What happens if you redefine the same path later in the same template but assign a value with a different primitive type? What happens if you define the path to the special literal `undef` between those two assignments?

5. What happens if you define the `/hardware/cpu/number` to be a boolean or string value?

6. Can you delete a property or resource that was previously defined in the template? Hint: try using the special literal `null`.

7. What happens if you try to set a path that does not begin with a slash?

5. Simple Type Checking

If you worked through the exercises of the previous section, you will have discovered that although you have an intuitive idea of what type a particular path should contain (e.g. `/hardware/cpu/number` should be positive long), the pan compiler does not. Downstream tools to configure a machine will likely expect certain values to have certain types and will produce errors or erroneous configurations if the correct type is not used. One of the strengths of the pan language is to specify constraints on the values to detect problems before configurations are deployed to machines.

5.1. Binding Primitive Types to Paths

At the most basic level, a system administrator can tell the pan compiler that a particular element must be a particular type. This is done with the `bind` statement. To tell the compiler that the path `/hardware/cpu/number` must be a long value, add the following statement to the `nfsserver.example.org` example.

```
bind '/hardware/cpu/number' = long;
```

This statement can appear anywhere in the file; all of the specified constraints will be verified *after* the complete configuration is built. Setting this path to a value that is not a long or not setting the value at all will cause the compilation to fail.

The above constraint only does part of the work though; the value could still be set to zero or a negative value without having the compiler complain. Pan also allows a range to be specified for primitive values. Changing the statement to the following:

```
bind '/hardware/cpu/number' = long(1..);
```

will require that the value be a positive long value. A valid range can have the minimum value, maximum value, or both specified. A range is always *inclusive* of the endpoint values. The endpoint values must be long literal values. A range specified as a single value indicates an exact match (e.g. `3` is shorthand for `3..3`). A range can be applied to a long, double, or string value. For strings, the range is applied to the length of the string. A range cannot be applied to boolean values.

5.2. User-Defined Types

While one could imagine adding bind statements for every named element within a configuration tree, this would quickly become tedious. To avoid unnecessary duplication, pan allows user-defined types both for properties and resources. One could define, for example, a type for a port number and then privileged and unprivileged ports:

```
type port = long(0..65535);
type priv_port = port(..1024);
type unpriv_port = port(1025..);
```

The above statements would define three new types: port, priv_port, and unpriv_port. Note that once a type has been defined, it may be used anywhere in the pan language that native types are allowed. Here the privileged and unprivileged port types are defined in term of port. The order is significant here; a valid type definition can only reference types that have been previously defined. Defining "alias"

types like this can reduce errors by enforcing consistent constraints and improve the readability of the code if appropriate names are chosen.

User-defined types can be bound to a path using exactly the same syntax as for binding with primitive types. Once again order is significant; referenced types must be defined before a `bind` statement is executed.

Homogeneous resources can be easily defined by adding brackets or braces to a type definition:

```
type port_list = port[];
type port_nlist = port{};
```

where these define a list or nlist, respectively. Empty brackets or braces means that there are no restrictions on the size of the resource; the resources may also be empty. If the brackets or braces contain a range, the constraint is applied to the size of the resource. For example, `long[3]` would be a long list with exactly three children and `long[1..]` would be a long list with one or more children.

One of the most common user-defined types is that of a *record*. A record is an nlist with specific, named children defined. For example, one can specify a disk_info type to define the disk information in the `nfsserver.example.org` example:

```
type disk = {
  'label' ? string
  'capacity' : long(1..)
  'boot' : boolean
};
```

This type definition would be an nlist with three children named `label`, `capacity`, and `boot`. The `label` is an optional child (indicated by the question mark) and may or may not exist. However, if it does exist then the given type constraint must be met. The other two are required children and they must exist for the type definition to validate correctly. Children with other names are not permitted by this type definition.

Occasionally it is useful to define an open or extensible record that allows children not specified in the type definition to exist. Imagine that for the machine location that the `rack` and `slot` information is required, but other information could be added as well. The following record definition:

```
type location = extensible {
  'rack' : string
  'slot' : long(0..50)
};
```

would allow someone to add information like the building number without violating this type's constraints.

5.3. Schema Definition

Using type definitions for properties and resources, one can build a global schema for a configuration. Using a global schema is extremely important when using pan to define configurations because it allows extensive compile-time checking of the configuration, avoiding having to cleanup the mess that can result from deploying a bad configuration. The following revisits the `nfsserver.example.org` configuration by adding a full schema:

```
object template nfsserver.example.org;

type location = extensible {
  'rack' : string
  'slot' : long(0..50)
};

type cpu = {
```

```
  'model' : string
  'speed' : double(0..)
  'arch' : string
  'cores' : long(1..)
  'number' : long(1..)
};

type disk = {
  'label' ? string
  'capacity' : long(1..)
  'boot' : boolean
};

type disks = {
  'ide' ? disk[]
  'scsi' ? disk{}
};

type nic = {
  'mac' : string
  'pxeboot' : boolean
};

type hardware = {
  'location' : location
  'ram' : long(0..)
  'cpu' : cpu
  'disks' : disks
  'nic' : nic[]
};

type root = {
  'hardware' : hardware
};

bind '/' = root;

'/hardware/location/rack' = 'IBM04';
'/hardware/location/slot' = 25;

'/hardware/ram' = 2048;

'/hardware/cpu/model' = 'Intel Xeon';
'/hardware/cpu/speed' = 2.5;
'/hardware/cpu/arch' = 'x86_64';
'/hardware/cpu/cores' = 4;
'/hardware/cpu/number' = 2;

'/hardware/disk/ide/0/capacity' = 64;
'/hardware/disk/ide/0/boot' = true;
'/hardware/disk/ide/0/label' = 'system';
'/hardware/disk/ide/1/capacity' = 1024;
'/hardware/disk/ide/1/boot' = false;

'/hardware/nic/0/mac' = '01:23:45:ab:cd:99';
'/hardware/nic/0/pxeboot' = false;
'/hardware/nic/1/mac' = '01:23:45:ab:cd:00';
'/hardware/nic/1/pxeboot' = true;
```

The series of type definitions progressively define the full schema from the lowest to highest levels. Note that there is only one `bind` statement in this configuration. This `bind` statement binds the root resource, specified by the path `'/'`, to the root of the schema. All of the other definitions are bound implicitly through this one statement.

5.4. Exercises

1. Determine what happens if the type definitions are specified in the wrong order.

2. Verify that the type definitions are fully applied to the `nfsserver.example.org` example by specifying some invalid values.

3. Also verify that adding paths outside of the schema is caught by the validation.

4. Although the schema in the last `nfsserver.example.org` template is much more restrictive, there are still several places where simple typos or inconsistent values could produce an invalid configuration. Where are they? What would you want to do to fix them?

5. What happens if you bind several different types to the same path?

6. Modular Configuration

6.1. Include Statement

So far only the hardware configuration and schema for one machine has been defined with the `nfsserver.example.org` configuration. One could imagine just doing a cut and paste to create the other three machines in our scenario. While this will work, the global site configuration will quickly become unwieldy and error-prone. In particular the schema is something that should be shared between all or many machines on a site. Multiple copies means multiple copies to keep up-to-date and multiple chances to introduce errors.

To encourage reuse of the configuration and to reduce maintenance effort, pan allows one template to include another (with some limitations). For example, the above schema can be pulled into another template (named `common/schema.tpl`) and included in the main object template.

```
declaration template common/schema;

type location = extensible {
  'rack' : string
  'slot' : long(0..50)
};

type cpu = {
  'model' : string
  'speed' : double(0..)
  'arch' : string
  'cores' : long(1..)
  'number' : long(1..)
};

type disk = {
  'label' ? string
  'capacity' : long(1..)
  'boot' : boolean
};
```

```
type disks = {
  'ide' ? disk[]
  'scsi' ? disk{}
};

type nic = {
  'mac' : string
  'pxeboot' : boolean
};

type hardware = {
  'location' : location
  'ram' : long(0..)
  'cpu' : cpu
  'disks' : disks
  'nic' : nic[]
};

type root = {
  'hardware' : hardware
};
```

The main object template then becomes:

```
object template nfsserver.example.org;

include { 'common/schema' };

bind '/' = root;

'/hardware/location/rack' = 'IBM04';
'/hardware/location/slot' = 25;

'/hardware/ram' = 2048;

'/hardware/cpu/model' = 'Intel Xeon';
'/hardware/cpu/speed' = 2.5;
'/hardware/cpu/arch' = 'x86_64';
'/hardware/cpu/cores' = 4;
'/hardware/cpu/number' = 2;

'/hardware/disk/ide/0/capacity' = 64;
'/hardware/disk/ide/0/boot' = true;
'/hardware/disk/ide/0/label' = 'system';
'/hardware/disk/ide/1/capacity' = 1024;
'/hardware/disk/ide/1/boot' = false;

'/hardware/nic/0/mac' = '01:23:45:ab:cd:99';
'/hardware/nic/0/pxeboot' = false;
'/hardware/nic/1/mac' = '01:23:45:ab:cd:00';
'/hardware/nic/1/pxeboot' = true;
```

There are three important changes to point out.

First, there is a new pan statement in the `nfsserver.example.org` template to include the schema. The `include` statement takes the name of the template to include as a string; the braces are mandatory. If the template is not included directly on the command line, then the compiler will search the *loadpath* for the template. If the loadpath is not specified, then it defaults to the current working directory.

Second, the schema has been pulled out into a separate file. The first line of that schema template is now marked as a `declaration` template. Such a template can only include type declarations. (Also variable and function declarations as we will see later.) Such a template will be included at most once when building an object; all inclusions after the first will be ignored. This allows many different template to reference type (and function) declarations that they use without having to worry about accidentally redefining them.

Third, the schema template name is `common/schema` and must be located in a file called `common/schema.tpl`; that is, it must be in a subdirectory of the current directory called `common`. This is called *namespacing* and allows the templates that make up a configuration to be organized into subdirectories. For the few templates that are used in this tutorial, namespacing is not critical. It is, however, critical for real sites that are likely to have hundreds or thousands of templates. Note that the hierarchy for namespaces is completely independent of the hierarchy used in the configuration schema.

Pulling out common declarations and help maintain coherence between different managed machines and reduce the overall size of the configuration. There are however, more mechanisms to reduce duplication.

6.2. Structure Templates

Sites usually buy many identical machines in a single purchase, so much of the hardware configuration for those machines is the same. Another mechanism that can be exploited to reuse configuration parameters is a `structure` template. Such a template defines an nlist that is initially independent of the configuration tree itself. For our scenario, let us assume that the four machines have identical RAM, CPU, and disk configurations; the NIC and location information is different for each machine. The following template pulls out the common information into a `structure` template:

```
structure template common/machine/ibm-server-model-123;

'ram' = 2048;

'cpu/model' = 'Intel Xeon';
'cpu/speed' = 2.5;
'cpu/arch' = 'x86_64';
'cpu/cores' = 4;
'cpu/number' = 2;

'disk/ide/0/capacity' = 64;
'disk/ide/0/boot' = true;
'disk/ide/0/label' = 'system';
'disk/ide/1/capacity' = 1024;
'disk/ide/1/boot' = false;

'location' = undef;
'nic' = undef;
```

The structure template is not rooted into the configuration (yet) and hence all of the paths in the assignment statements must be *relative*; that is, they do not begin with a slash. Also, the `location` and `nic` children were set to `undef`. These are the values that will vary from machine to machine, but we want to ensure that anyone using this template sets those values. If someone uses this template, but forgets to set those values, the compiler will abort the compilation with an error. The `undef` value may not appear in a final configuration.

How is this used in the machine configuration? The `include` statement will not work because we must indicate where the configuration should be rooted. The answer is to use an assignment statement along with the `create` function.

```
object template nfsserver.example.org;
```

```
include { 'common/schema' };

bind '/' = root;

'/hardware' = create('common/machine/ibm-server-model-123');

'/hardware/location/rack' = 'IBM04';
'/hardware/location/slot' = 25;

'/hardware/nic/0/mac' = '01:23:45:ab:cd:99';
'/hardware/nic/0/pxeboot' = false;
'/hardware/nic/1/mac' = '01:23:45:ab:cd:00';
'/hardware/nic/1/pxeboot' = true;
```

Finally, the machine configuration contains only values that depend on the machine itself with common values pulled in from shared templates.

Although the example here uses the hardware configuration, in reality it can be used for any subtree that is invariant or nearly-invariant. One can even reuse the same structure template many times in the same object just be creating a new instance and assigning it to a particular part of the tree.

6.3. Exercises

1. What happens if you put an absolute assignment statement in a structure template?

2. What happens if you put a relative assignment statement in an object template?

3. Come up with an example where you might want to reuse the same structure template several times in the same object.

4. What happens if you try to include an object template from another object, declaration, or structure template?

5. What happens if you try to include a structure template with an include statement?

7. Default Values

Looking again at the nfsserver.example.org configuration, there are a couple of places where we could hope to use default values. The `pxeboot` and `boot` flags in the nic and disk type definitions could use default values. In both cases, at most one value will be set to `true`; all other values will be set to `false`. Another place one might want to use default values is in the cpu type; perhaps we would like to have `number` and `cores` both default to 1 if not specified.

Pan allows type definitions to contain default values. For example, to change the three type definitions mentioned above:

```
type cpu = {
  'model' : string
  'speed' : double(0..)
  'arch' : string
  'cores' : long(1..) = 1
  'number' : long(1..) = 1
};

type nic = {
  'mac' : string
```

```
  'pxeboot' : boolean = false
};

type disk = {
  'label' ? string
  'capacity' : long(1..)
  'boot' : boolean = false
};
```

With these definitions, the lines which set the `pxeboot` and `boot` flags to false can be removed from the configuration and the compiler will still produce the same result. The default value will only be used if the corresponding element does not exist or has the `undef` value *after all* of the statements for an object have been executed. Consequently, a value that has been explicitly defined will always be used in preference to the default. Although one can set a default value for an optional field in a record, it will have an effect *only* if the value was explicitly set to `undef`.

The default values must be a compile time constants.

7.1. Exercises

1. Update the schema template using the type definitions with defaults. Remove the unnecessary lines in the `nfsserver.example.org` template and ensure that the defaults are correctly used.

2. Can you set a default value to an illegal value? When is the illegal value detected?

3. Can you set a default value to `undef`?

4. Set a default value for an optional field and see if/when this is used?

8. Data Manipulation Language

Although a declarative language has many benefits, there are times when values need to be calculated or verified based on some algorithm. To allow this in the pan language without giving up the fundamental declarative nature of the language, pan allows a Data Manipulation Language (DML) block to appear in most places where a literal value can appear. The DML block can use information contained in global variables, in the current object, or in other objects to calculate a value. This makes the language much more flexible without giving up the simplicity of the overall pan syntax.

A DML block is a sequence of one or more expressions, where every expression returns a value. The value for an entire DML block is the value of the last executed statement. The DML syntax looks like a simplified version of c. The DML block can only return a value and cannot directly or indirectly change the configuration tree nor the global state except by assigning the return value to a path or variable.

8.1. Operators

DML has a complete set of operators for arithmetic, bit operations, and comparison operators. These are identical to those available in Java, including that the plus operator (+) works also to concatenate strings. See the pan language reference for a complete list of all of the operators. Where necessary in a calculation, DML will promote long values to doubles; no other automatic conversions are performed.

```
'/result1' = 147 + 10;  # will be 157
'/result2' = 153 == (150 + 3);  # will be true
'/result3' = {
  5 + 5;
  20 % 5;
  6 * 5.0;
```

```
};  # will be 30.0
```

In assignment statements, single expressions can be used after the equal sign without brackets. DML blocks with multiple statements must be surrounded by braces. DML blocks may appear as default values in type definitions if the block evaluates to a constant, compile-time value.

8.2. Variables

There are three types of variables in the pan language. Local variables defined and used within a DML block, global variables defined by the `variable` statement and automatic variables provided by the compiler.

Local variables can be defined within a DML block simply by assigning a value to them. There is no need to declare the variable before assigning a value; the variable must have been defined before using it on the right-side of an expression.

```
'/result' = {
  x = 10;
  y = 20;
  z = 30;
  x + y - z;
};  # will be zero
```

Once the DML block has finished, all of the local variables are destroyed and are no longer accessible.

Global variables can be defined by using the pan `variable` statement. For example,

```
variable X = 'hello world!';
```

will define a global variable `X` that will be accessible from any DML block evaluated after this `variable` statement is executed. The variable definition is available until the end of the validation phase. Note that global variables are global with respect the the object being compiled. Each object has its own variable table; thus global variables cannot be used to transmit information between objects. To avoid naming conflicts between global and local variables, it is best practice to use all capital letters for global variable names.

Automatic variables are provided by the compiler in certain contexts. The automatic variables are `SELF`, `OBJECT`, `FUNCTION`, `ARGC`, and `ARGV`. The variable `OBJECT` is the easiest to explain; it contains the name of the object currently being executed.

```
object template dummy.example.org;
'/name' = OBJECT;
```

The path will contain the string "dummy.example.org" after execution. As the object templates are usually named either directly or indirectly after the machines hostname, `OBJECT` is often used to lookup host-specific information. The `FUNCTION` variable contains the name of the current function, often used for debugging or error statements. The `ARGC` and `ARGV` variables are defined only within a user-defined function and correspond to the number of arguments and a list of those arguments, respectively.

In an assignment statement, `SELF` contains the value of the path before the DML block started executing. Within a DML block, one cannot directly change the value of `SELF` but one can indirectly change the value by assigning values to children of `SELF`. This is best illustrated by an example:

```
object template self-test;

'/result/a' = 'value a is set';

'/result' = {
```

```
  SELF['b'] = 'value b is set';
  SELF;
};

$ panc --xml-style=text self-test.tpl
$ cat self-test.txt
+-profile
        +-result
                $ a : (string) 'value a is set'
                $ b : (string) 'value b is set'
```

The compiler optimizes references to the SELF variable, so making incremental changes to a resource like this is recommended. One common error when using SELF is not to remember to return SELF as the last expression in a block; this can lead to unexpected results or errors.

Any type can define a DML block to act as a validation function during the validation phase. The DML block in this case must return `true`, if the validation was OK, or `false`, otherwise. In such a validation block, the SELF variable is assigned the value of the element being validated. No changes whatsoever can be made to the configuration tree or to the SELF variable during the validation phase.

8.3. Functions

There are a large number of built-in functions available; see the pan language reference for the complete list. The most commonly used functions are given in the following table.

`nlist`(key, value, ...)	create an nlist from the given parameters
`list`(value, ...)	create a list from the given parameters
`create`(name, key, value, ...)	create an nlist from the named structure template and provided key/value pairs
`length`(resource), `length`(string)	returns the length of the given resource or string
`match`(regexp, string)	return boolean indicating if the given string matches the given regular expression
`matches`(regexp, string)	return a list of matched groups based on the given regular expression and string
`to_string`(value), `to_long`(value), `to_boolean`(value), `to_double`(value)	type conversion functions
`is_string`(value), `is_list`(value), `is_nlist`(value), ...	type checking functions return a boolean if the argument is of the indicated type
`debug`(message), `traceback`(message), `error`(message)	debugging functions; `debug` and `traceback` will return `undef`; `error` will abort the processing
value(path)	return the value at the named path; the path may be an absolute or external path

In addition to the built-in functions, users can also define functions. The `function` statement allows any DML block to be treated as a function. After the function is defined, it may be called within any subsequent block just as a built-in function could. For example, the following will take the average of a sequence of numbers.

```
object template function-test;

function average = {

  # Can't take an average without at least one value.
```

```
  if (ARGC == 0) {
    error('at least one argument must be given to average()');
  };

  # Iterate over all of the values and keep running sum.
  sum = 0.0;
  foreach (key; value; ARGV) {

    # Ensure that the argument is a number.
    if (! is_number(value) ) {
      error('non-numeric value encountered: ' + to_string(value));
    };
    sum = sum + value;
  };

  # Now give back the average.
  sum/ARGC;
};

'/result' = average(1, 2.5, 3, 5);
```

```
$ panc --xml-style=text function-test.tpl
$ cat function-test.txt
+-profile
        $ result : (double) '2.875'
```

Functions are commonly used to perform some common algorithm or to do some validation. If the function is used for validation, then the function must return a boolean value. Note that all variables within the scope of a function are local to the function and cannot influence (or be influenced by) variables of the same name at other levels of the call stack.

8.4. Flow Control

In the DML, there are three expressions that can alter the normal sequential flow of execution: an `if` expression for branching, a `while` expression for looping, and a `foreach` expression for iteration. All of these are expressions that return a value. The value of each expression is the value of the last expression evaluated in the `if`, `while`, or `foreach` block. If the block is never executed, then the expression returns `undef`.

The syntax for the `if` and `while` expressions are the same as in most programming languages. The syntax for the `foreach` expression requires an example:

```
object template foreach-test;

'/nlist-result' = {
  result = '';

  x = nlist('a', 1, 'b', 2, 'c', 3);

  foreach (key; value; x) {
    result = result + key + " -> " + to_string(value) + "    ";
  };

  result;
};

'/list-result' = {
  result = '';
```

```
  y = list('alpha', 'beta', 'gamma');

  foreach (key; value; y) {
    result = result + to_string(key) + " -> " + value + "    ";
  };

  result;
};
```

```
$ panc --xml-style=text foreach-test.tpl
$ cat foreach-test.txt
+-profile
        $ list-result : (string) '0 -> alpha    1 -> beta    2 -> gamma    '
        $ nlist-result : (string) 'a -> 1    b -> 2    c -> 3    '
```

The `foreach` expression iterates over all of the children of the given resource, `x` and `y` in this example. For each child, the iteration variables (`key` and `value` here) are assigned the key/index and value of the child. Any valid names can be used for the iteration variables.

8.5. Exercises

1. Define a function that pushes a value on the end of a list. What happens if the list does not exist?

2. Define a function that pushes a value on the end of `SELF`. What happens if the list does not exist? If the function does not work in this case, can you create a version that does?

9. Software Configuration

On a machine managed by quattor, a daemon accepts notifications of configuration changes and then runs a set of "components" to affect those changes on the client. A component is simply a script that reads configuration information from the machine profile and then makes appropriate changes to the services running on the client. This includes starting, stopping, or restarting services as appropriate. Although it would be possible to create a single component that handles all of the machine configuration, this is neither scalable nor maintainable. Usually, a single component is responsible for a single, low-level service.

In our scenario, there will be four components to configure: NFS, Torque, users, and firewall. The author of a component defines an appropriate configuration schema for the service and provides that schema via a pan language template. System administrators can then use that schema to set the service parameters via the pan language.

9.1. Components

Components are built on a common skeleton and share some common parameters. Our reduced example will have an flag to indicate if a component is active and lists of dependencies. An overly simplistic schema might be the following:

```
type component = extensible {
  active : boolean = true
  pre ? string[]
  post ? string[]
};
```

A series of components could be bound to a particular part of the configuration tree. The usual schema used with quattor puts the components at `/software/components/` as an nlist; the bind statement to accomplish this is:

```
bind '/software/components' = component{};
```

Each component is identified by a unique key in this nlist.

The `extensible` keyword on the record definition is extremely important. We expect each component to define additional parameters specific to the service it treats. Without the `extensible` keyword, the pan compiler would only allow the three children that are explicitly defined in the component type definition; the `extensible` keyword allows other children to exist.

The above definition is overly simplistic because it does not validate the component's values very well. The active flag is fine; however, the pre- and post-dependencies should be limited to other active components specified in the configuration. Similarly, any additional parameters should be validated as much as possible to avoid having invalid parameters used to configure services. The next chapter will concentrate on the more advanced validation features available in the pan language.

9.2. NFS Component Schema

The Network File System (NFS) is a service that allows a machine to export certain paths that can then be mounted by other machines within their file systems. Correspondingly, there are two parts of the NFS configuration: one part for a server that exports paths and one part for the client that mounts those remote file systems. Note that any given machine can be a client, server, or both. The following schema captures the parameters needed for NFS configuration:

```
# Type that defines path and authorized host for NFS server export.
type nfs_exports = {
  'path' : string
  'authorized_host' : string
};

# Type containing parameters to mount remote NFS volume.
type nfs_mounts = {
  'host' : string
  'path' : string
  'mountpoint' : string
};

# Allows lists of NFS exports and NFS mounts (both optional).
type config_nfs = {
  include component
  'exports' ? nfs_exports[]
  'mounts'  ? nfs_mounts[]
};
```

Assuming the component type has been defined and bound to `/software/components` as an nlist and the config_nfs type has been defined, the following will ensure that the `/software/components/nfs` path meets both the component and config_nfs types:

```
bind '/software/components/nfs' = config_nfs;
```

Note that with the above bind, this implies that this path has two type definitions associated with it: component and config_nfs. The pan language allows multiple types to be defined to a path and enforces all of them. That is, all of the types bound to a path must be valid for the configuration as a whole to be valid. Be careful not to assign multiple incompatible types to the same path. For example, the following will never lead to a validated configuration:

```
object template never-valid;

bind '/result' = string;
bind '/result' = boolean;
```

```
'/result' = 'OK';
```

because either one or the other of the bound types will fail. Real-world conflicts of this type are usually more complicated but, at the most basic level, arise because of incompatible primitive types being assigned to the same path.

As can be seen above, allowing multiple types to be bound to the same path permits pan to have functionality similar to object inheritance. In reality, this is a "duck" typing system that simply checks that all of the bound types are simultaneously satisfied.

9.3. Torque Component Schema

Torque is a commonly-used batch system. Like the NFS configuration, there are client and server aspects to running Torque. One difference, however, is that a client can be associated with only one server. The server configuration consists of a set of queues and list of client nodes associated with the server. The client configuration simply indicates if the client's state and the name of the server. The schema is:

```
# Torque server information.
type torque_server = {
  'queues' : string[1..]
  'workers' ? string[]
};

# Torque client information.
type torque_client = {
  'server' : string
  'state' : string with match(SELF, 'open|closed|drain');
};

# Overall configuration.
type config_torque = {
  include component
  'server_params' ? torque_server
  'client_params' ? torque_client
};
```

This can be bound to part of the configuration schema like was done for NFS. In our example, will will use the path `/system/components/torque`.

9.4. Schemas for Users and a Firewall

In a complete system there are a large number of services that need to be configured. Many of these services share common needs for low-level configuration of things like the users and open ports on the firewall. These are included just to show some of the best practices when organizing configuration templates. The simplified schemas are:

```
# Simple user configuration component.
type config_users = {
  include component
  'uid' : long(0..){}
};

# Simple firewall configuration component.
type config_firewall = {
  include component
  'open' : long(0..)[]
```

```
};
```

This allows usernames to be associated to a given UID and to specify open ports in the firewall.

9.5. Component Configuration Organization

Pan puts very few constraints on the organization of the configuration information for components. Nonetheless, some best practices have arisen with use of the system and now almost every component exposes two pan templates: `schema.tpl` and `config.tpl`. The schema template contains a component's schema definition and any associated validation functions. The other file contains the default configuration for the component, such as standard dependencies, global variables, etc. The schema file for Torque might look like the following:

```
declaration template component/torque/schema;

include { 'quattor/structure_component' };

# Torque server information.
type torque_server = {
  'queues' : string[1..]
  'workers' ? string[]
};

# Torque client information.
type torque_client = {
  'server' : string
  'state' : string with match(SELF, 'open|closed|drain');
};

# Overall configuration.
type config_torque = {
  include component
  'torque_user' ? string
  'server_params' ? torque_server
  'client_params' ? torque_client
};
```

and the associate default configuration file:

```
unique template component/torque/config;

# Define the schema for the Torque configuration.
include { 'component/torque/schema' };

# Define some default port numbers.
variable TORQUE_CLIENT_PORT ?= 9999;
variable TORQUE_SERVER_port ?= 9998;

# Bind the schema to a particular place in the configuration tree.
bind '/software/components/torque' = config_torque;

# Set the component to be active by default.
'/software/components/torque/active' ?= true;
```

To ensure that the necessary schema is defined and to ensure that all of the default actions have been taken, it is wise to always include the `config.tpl` file before each block of Torque configuration statements. Notice that these templates are defined as a `declaration` and `unique` templates, respectively, to avoid any performance penalty for including these files repeatedly. Usually a system administrator would only include the `config.tpl` file directly.

Distributions of templates usually go one step further and define templates for common service configurations. For example, one could create a Torque server that does the default configuration for a machine running a Torque server. Take the following example:

```
unique template service/torque/server;

include { 'component/torque/config' };

# Change the default port, setup the queues, and identify the worker nodes.
variable TORQUE_SERVER_PORT = 1212;
'/software/components/torque/server_params/queues' = list('short', 'medium', 'l
'/software/components/torque/server_params/workers' = list('worker1.example.org

# Open the correct port on the firewall.
include { 'component/firewall/config' };
'/software/components/firewall/open' = {
  SELF[length(SELF)] = TORQUE_SERVER_PORT;
};

# Setup the user for torque.
include { 'component/users/config' };
'/software/components/users/uid/torque_mgr' = 1000;
```

This template would do all that is necessary to configure a Torque server, including the configuration of the low-level services. A system administrator wanting to use this, would then create an object template like the following:

```
object template torque-server;

# ... some machine hardware configuration ...

# Run a Torque server on this machine.
include { 'service/torque/config' };

# ... inclusion of other high-level service configurations ...
```

Using the conventions described above allows maximum reuse of the configuration information and makes it easy to mix-and-match high-level services for a particular object template.

10. General Validation

The greatest strength of the pan language is the ability to do detailed validation of configuration parameters, of correlated parameters within a machine profile, and of correlated parameters *between* machine profiles. Although the validation can make it difficult to get a particular machine profile to compile, the time spent getting a valid machine configuration before deployment more than makes up for the time wasted debugging a bad configuration that has been deployed.

Simple validation through the validation of primitive properties and simple resources has already been covered when discussing the pan type definition features. This chapter deals with more complicated scenarios.

10.1. Advanced Parameter Validation

Often there are cases where the legal values of a parameter cannot be expressed as a simple range. The pan language allows you to attach arbitrary validation code to a type definition. The code is attached to the type definition using the `with` keyword. Consider the following examples:

```
type even_positive_long = long(1..) with (SELF % 2 == 0);
type machine_state_enum = string with match(SELF, 'open|closed|drain');
```

```
type ip = string with is_ipv4(SELF);
```

The validation code must return the boolean value `true`, if the associated value is correct. Returning any other value or raising an error with the `error` function will cause the build of the machine configuration to abort.

Simple constraints are often written directly with the type statement; more complicated validation usually calls a separate function. The third line in the example above calls the function `is_ipv4`. This is a user-defined function that could look like:

```
function is_ipv4 = {
  terms = split('\.', ARGV[0]);
  foreach (index; term; terms) {
    i = to_long(term);
    if (i < 0 || i > 255) {
      return(false);
    };
  };
  true;
};
```

A real version of this function would probably do a great deal more checking of the value and probably raise errors with more intuitive error messages.

10.2. Validation of Correlated Configuration Parameters

Often the correct configuration of a machine requires that configuration parameters in different parts of the configuration are correlated. One example is the validation of the pre- and post-dependencies of the component configuration. It makes no sense for one component to depend on another one that is not defined in the configuration or is not active.

The following validation function accomplishes such a check, assuming that the components are bound to `/software/components`:

```
function valid_component_list = {

  # ARGV[0] should be the list to check.

  # Check that each referenced component exists.
  foreach (k; v; ARGV[0]) {

    # Path to the root of the named component.
    path = '/software/components/' + v;

    if (!exists(path)) {
      error(path + ' does not exist');
    } else {

      # Path to the active flag for the named component.
      active_path = path + '/active';

      if (!(is_defined(active_path) && value(active_path))) {
        error('component ' + v + ' isn't active');
      };

    };
```

```
  };

};

type component_list = string[] with valid_component_list(SELF);

type component = extensible {
  active : boolean = true
  pre ? component_list
  post ? component_list
};
```

It also defines a component_list type and uses this for a better definition of a the component type. This will get run on anything that is bound to the component type, directly or indirectly. Note how the function looks at other values in the configuration by creating the path and looking up the values with the `value` function.

The above function works but has one disadvantage: it will only work for components defined below `/software/components`. If the list of components is defined elsewhere, then this schema definition will have to be modified. One can usually avoid this by applying the validation to a common parent. In this case, we can add the validation to the parent.

```
function valid_component_nlist = {

  # Loop over each component.
  foreach (name; component; SELF) {

    if (exists(component['pre'])) {
      foreach (index; dependency; component['pre']) {
        if (!exists(SELF['dependency']['active'] ||
             SELF['dependency']['active'])) {
          error('non-existant or inactive dependency: ' + dependency);
        };
      };
    };

    # ... same for post ...

  };

};


type component = extensible {
  active : boolean = true;
  pre ? string[]
  post ? string[]
};

type component_nlist = component{} with valid_component_nlist(SELF);
```

This will accomplish the same validation, but will be independent of the location in the tree. It is, however, significantly more complicated to write and to understand the validation function. In the real world, the added complexity must be weighed against the likelihood that the type will be re-located within the configuration tree.

The situation often arises that you want to validate a parameter against other siblings in the machine configuration tree. In this case, we wanted to ensure that other components were properly configured; to know that we needed to search "up and over" in the machine configuration. The pan language does

not allow use of relative paths for the `value` function, so the two options are those presented here. Use an absolute path and reconstruct the paths or put the validation on a common parent.

10.3. Cross-Machine Validation

Another common situation is the need to validate machine configurations against each other. This often arises in client/server situations. For NFS, for instance, one would probably like to verify that a network share mounted on a client is actually exported by the server. The following example will do this:

```
# Determine that a given mounted network share is actually
# exported by the server.
function valid_export = {

  info = ARGV[0];
  myhost = info['host'];
  mypath = info['path'];

  exports_path = host + ':/software/components/nfs/exports';

  found = false;
  if (path_exists(exports_path)) {

    exports = value(exports_path);

    foreach (index; einfo; exports) {
      if (einfo['authorized_host'] == myhost &&
          einfo['path'] == mypath) {
        found = true;
      };
    };

  };
  found;
};

# Type that defines path and authorized host for NFS server export.
type nfs_exports = {
  'path' : string
  'authorized_host' : string
};

# Type containing parameters to mount remote NFS volume.
type nfs_mounts = {
  'host' : string
  'path' : string
  'mountpoint' : string
} with valid_export(SELF);

# Allows lists of NFS exports and NFS mounts (both optional).
type config_nfs = {
  include component
  'exports' ? nfs_exports[]
  'mounts'  ? nfs_mounts[]
};
```

To do this type of validation, the full external path must be constructed for the `value` function. This has the same disadvantage as above in that if the schema is changed the function definition needs to

be altered accordingly. The above code also assumes that the machine profile names are equivalent to the hostname. If another convention is being used, then the hostname will have to be converted to the corresponding machine name.

It is worth noting that all of the validation is done *after* the machine configuration trees are built. This allows circular validation dependencies to be supported. That is, clients can check that they are properly included in the server configuration and the server can check that its clients are configured. A batch system is a typical example where this circular cross-validation is useful.

10.4. Exercises

1. Devise a schema for a client/server system that you are familiar with.
2. Add validation for all of the parameters.
3. Create the `config.tpl` and `schema.tpl` files for this system.
4. Create client and server files for this service. Incorporate them in object templates and ensure that they behave as expected.
5. Add validation to verify the client and server machine configurations against each other.
6. Create the validation functions necessary to do the circular cross-validation described for the Torque batch system.

11. Conclusions

This tutorial covered the highlights of the pan language. After finishing this tutorial, you should be able to create and maintain a site configuration written in the pan language. This tutorial, however, did not cover any of the configuration conventions of a particular community. Ask others in the community about common conventions (e.g. on the Quattor mailing list or the Quattor Working Group (QWG) wiki); links to other sources of information can be found on the Quattor web site. You can also find more information about the pan language and the pan compiler in the other documents distributed with the compiler.