Pan Configuration Language

9.2

Charles Loomis

Pan Configuration Language: 9.2

Charles Loomis

Publication date 2012-02-20

Table of Contents

List of Figures

List of Tables

Preface

Organization

This book is intended to act as both a reference guide for the pan configuration language as well as a tutorial on using the associated compiler. The first chapter introduces the language and guides you through a basic installation of the compiler. The following chapter provides a simplified, real-world example to show the major features of the pan language for site configuration. Chapters 3-8 provide a detailed description of the pan language and act as a reference for it. Chapters 9-11 provide information about advanced features and best practices when using the language. Finally, Chapter 12 gives some information about troubleshooting problems that can arise when using the compiler and language. The appendices provide detailed information on installing and using the compiler in various environments as well as detailed information on the pan commands and functions.

Typographic Conventions

Table 1. Typographic Conventions

`filename`	References to files are typeset in this style. In this book, these are usually references to configuration templates.
command	Commands to be executed from the command line are typeset in this style. This is usually a direct or indirect invocation of the pan configuration language compiler.
`keyword`	Pan configuration language keywords are typeset in this style. They represent the language's reserved words and should appear in configuration files exactly as written.

CHAPTER 1

Getting Started

The pan configuration language allows system administrators to define simultaneously a site configuration and a schema for validation. As a core component of the Quattor fabric management toolkit, the pan compiler translates this high-level site configuration to a machine-readable representation, which other tools can then use to enact the desired configuration changes.

Configuration Language

The pan language was designed to have a simple, human-friendly syntax. In addition, it allows more rigorous validation via its flexible data typing features when compared to, for instance, XML and 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 the templates are executed to generate a partial tree of configuration information. 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 any default values are inserted for missing elements. The tree of configuration information is complete after this stage.

validation	The configuration information is frozen and all standard and user-specified validation is done. Any invalid values or conditions will cause the processing to abort.
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. Usually there is one object template for each physical machine; although with the rise of virtualization, it may be one per logical machine.

Benefits

Using the pan language and compiler has the following benefits:

- Declarative language allows easier merging of configurations from different administrators.
- Encourages organization of configuration by service and function to allow sharing of configurations between machines and sites.
- Provides simple syntax for definition of configuration information and validation.
- Ensures a high-level of validation before configurations are deployed, avoiding interruptions in services and wasted time from recovery.

The language and compiler are intended to be used with other tools that manage the full set of configuration files and that can affect the changes necessary to arrive at the desired configuration. The Quattor toolkit provides such tools, although the compiler can be easily used in conjunction with others.

Download and Installation

The pan compiler can be invoked via the Unix (Linux) command line, ant, or maven. The easiest for the simple examples in this book is the command line interface. (See Appendix A for installation instructions for all the execution methods.) Locate and download the latest version of the pan tarball and untar this into a convenient directory. You can find the packaged versions of the compiler in the Quattor project space on SourceForge.

The pan compiler requires a Java Runtime Environment (JRE) or Java Development Kit (JDK) 1.5 or later. If you will just be running a binary version of the pan compiler, the JRE is sufficient; compiling the sources will require the JDK. Use

a complete, certified version of the Java Virtual Machine; in particular avoid the GNU Java Compiler (GJC) as the pan compiler will not run correctly with it.

To use the compiler from the command line, you must make it accessible from the path.

```
$ export PANC_HOME=/panc/location
$ export PATH=$PATH:$PANC_HOME/bin
```

The above will work for Bourne shells; adjust the command for the shell that you use. Change the value of `PANC_HOME` to the directory where the pan compiler was unpacked.

Validating the Installation

Once you have installed the compiler, make sure that it is working correctly 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. If either command fails, review the installation instructions.

Invoking the Pan Compiler

Now create a file (called a "template") named `hello_world.pan` that contains the following:

```
object template hello_world;
'/message' = 'Hello World!';
```

Compile this template into the default XML representation and look at the output.

```
$ panc hello_world.pan

$ cat hello_world.xml
<?xml version="1.0" encoding="UTF-8"?>
<nlist format="pan" name="profile">
<string name="message">Hello World!</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 World!". 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 corresponds to the name of the field in the pan template.

The pan compiler can generate output in three additional formats: json, text, and dot. The following shows the output for the json format.

```
$ panc --xml-style=json hello_world.pan
```

```
$ cat hello_world.json
{
  "message": "Hello World!"
}
```

Warning

The alternate XML format "xmldb" is deprecated. This format is not compatible with arbitrary nlist keys. Use the 'pan' XML format for the JSON format instead.

In this book, the most convenient representation is the text format. This provides a clean representation of the configuration tree in plain text.

```
$ panc --xml-style=text hello_world.pan

$ cat hello_world.txt
+-profile
        $ message : (string) 'hello'
```

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

The last style is the "dot" format.

```
$ panc --xml-style=dot hello_world.pan

$ cat hello_world.dot
digraph "profile" {
bgcolor = beige
node [ color = black, shape = box, fontname=Helvetica ]
edge [ color = black ]
"/profile" [ label = "profile" ]
"/profile/message" [ label = "message\n'Hello World!'" ]
"/profile" -> "/profile/message"
}
```

Although the text is not very enlightening by itself, it can be used by Graphviz [http://www.graphviz.org/] to generate a graph of the configuration. Processing the above file with Graphviz produces the image shown in Figure 1.1, “Graph of configuration produced by `hello_world.pan`.”.

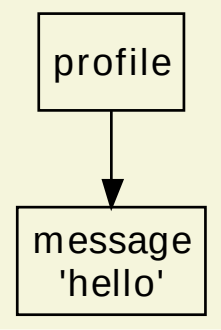

Figure 1.1. Graph of configuration produced by `hello_world.pan`.

CHAPTER 2

A Whirlwind Tour

This tour will highlight the major features of the pan language by showing how the configuration for a batch system for asynchronous handling of jobs could be described with the pan language. The fictitious, simplified batch system used here gives you the flavor of the development process and common pan features. The description of a real batch system would contain significantly more parameters and services.

Batch System Description

A batch system provides a set of resources for asynchronous execution of jobs (scripts) submitted by users. The batch system (or cluster) consists of:

Server (or head node)	A machine containing a service for accepting job requests from users and a scheduler for dispatching those jobs to available workers.
Workers	Machines that accept jobs from the server, execute them, and then return the results to the server.

Users send a script containing the job description to the server. The server then queues the request for later execution. The scheduler periodically checks the queued jobs and resources, sending a queued job for execution on a worker if one is available. The worker executes the job it has been given and keeps the server informed about the state of the job. At the end of the job, results are returned to the server. The user can interact with the server to determine the status of jobs and to retrieve the output of completed jobs.

For our simplified batch system, we want to create a set of parameters that describe the configuration. For many real services, the configuration schema used in pan will closely mirror the configuration file(s) of the service. In our case we will create a configuration schema based on the above description.

The server controls a set of workers and manages jobs via a set of queues. Each queue is named, has a CPU limit, and can be enabled or disabled. Each node also has a name, participates in one or more queues, and has a set of capabilities (e.g. a particular software license is available, has a fast network connection, etc.).

The worker needs to know with which server to communicate. Each worker will also have a flag to indicate if the worker is enabled or disabled.

Naive Configuration

Given the previous description, a pan language configuration for both the batch server and one batch worker can easily be created. We must create an object template for each machine in order to have the machine descriptions created during the compilation. Create the file `server.example.org.pan` with the following contents:

```
object template server.example.org;

'/batch/server/nodes/worker01.example.org/queues'
    = list('default');

'/batch/server/nodes/worker01.example.org/capabilities'
    = list('sw-license', 'fast-network');

'/batch/server/queues/default/maxCpuHours' = 1;
'/batch/server/queues/default/enabled' = true;
```

It is customary to use the machine name as the object template name. For this server, there is one worker node named 'worker01.example.org' and one queue named 'default'. The worker node participates in the 'default' queue and has a couple of capabilities. The 'default' queue has a CPU limit of 1 hour.

Create the file `worker01.example.org.pan` for the worker:

```
object template worker01.example.org;

'/batch/worker/server' = 'server.example.org';
'/batch/worker/enabled' = true;
```

This is part of the cluster controlled by the server 'server.example.org' and is enabled.

These templates can be compiled with the following command:

```
$ panc --xml-style=text *.pan
```

which then produces the files `server.example.org.txt` and `worker01.example.org.txt`:

```
+-profile
  +-batch
    +-server
      +-nodes
        +-worker01.example.org
```

```
          +-capabilities
            $ 0 : (string) 'sw-license'
            $ 1 : (string) 'fast-network'
          +-queues
            $ 0 : (string) 'default'
      +-queues
        +-default
          $ maxCpuHours : (long) '1'
```

```
+-profile
  +-batch
    +-worker
      $ enabled : (boolean) 'true'
      $ server : (string) 'server.example.org'
```

These generated files (or more likely their equivalents in XML) can then be used by tools to actually configure the machines and batch services appropriately.

Using Namespaces and Includes

The naive configuration shown in the previous section has a couple of problems. First, it will become tedious to maintain, especially if individual machines contain a mix of different services. Second, similar configurations would be duplicated between object templates, increasing the likelihood of errors. These problems can be eliminated by refactoring the configuration into separate templates and by organizing those templates into reasonable namespaces.

As a first step in reorganizing the configuration, we pull out the batch server and worker configurations into separate *ordinary* templates. These configurations are put into `services/batch-server.pan` and `services/batch-worker.pan`, respectively.

```
template services/batch-server;

'/batch/server/nodes/worker01.example.org/queues'
    = list('default');

'/batch/server/nodes/worker01.example.org/capabilities'
    = list('sw-license', 'fast-network');

'/batch/server/queues/default/maxCpuHours' = 1;
'/batch/server/queues/default/enabled' = true;
```

```
template services/batch-worker;

'/batch/worker/server' = 'server.example.org';
'/batch/worker/enabled' = true;
```

Note that these files are not object templates (i.e. there is no `object` modifier) and will not produce any output files themselves. Note also that they are namespaced; the relative directory of the template must match the path hierarchy in the file system. In this particular case, these both must appear in a `services` subdirectory.

Object templates can also be namespaced; here we will put them into a `profiles` subdirectory. These object templates can then include configuration in other (non-object) templates. The contents of these profiles becomes:

```
object template profiles/server.example.org;

include 'services/batch-server';
```

```
object template profiles/worker01.example.org;

include 'services/batch-worker';
```

Organizing the service configurations in this way makes it easy to include multiple services in a particular object template. If reasonable names are chosen, then the object template becomes self-documenting, listing the services included on the machine.

The command to compile these object templates is slightly different:

```
$ panc --xml-style=text profiles/*.pan
```

The output files by default will be placed next to the object template, so in this case they will be in the `profiles` subdirectory. You can verify that the reorganized configuration produces exactly the same configuration as the first example.

Simple Typing

Although the configuration is completely specified in the previous examples, it does not protect you from inappropriate values, for instance, specifying 'ON' for the boolean worker's *`enabled`* parameter or a negative number for the *`maxCpuHours`* parameter of a queue. The pan language has a number of primitive types, collections, and mechanisms for user-defined types.

Create a file named `services/batch-types.pan` with the following content:

```
declaration template services/batch-types;❶

type batch_capabilities❷ = string[];

type batch_queue_list❸ = string[1..];

type batch_node❹ = {
  'queues' : batch_queue_list
  'capabilities' ? batch_capabilities
};

type batch_queue❺ = {
  'maxCpuHours' : long(0..)
  'enabled' : boolean
};

type batch_server❻ = {
  'nodes' : batch_node{}
  'queues' : batch_queue{}
};

type batch_worker❼ = {
  'server' : string
  'enabled' : boolean
};
```

❼ The batch_worker type defines a record (nlist or hash with named children) for the worker configuration. The 'enabled' flag is defined to be a boolean value. The 'server' is defined to be a string. For a real configuration, the server would likely be define to be a hostname or IP address with appropriate constraints.

❻ The batch_server type also defines a record with nodes and queues children. These are both defined to be nlists where the keys are the worker host name or the queue name, respectively. The notation `mytype{}` defines an nlist.

❺ Type batch_queue type defines a record with the characteristics of a queue. Each queue can be enabled or disabled. The *`maxCpuHours`* is required to be a non-negative long value. The range specification `(0..)` limits the allowed values. Range limits like this apply to the numeric value for long and double types; it applies to the length for strings.

❹ Type batch_node again defines a record for a single node. The node description contains a list of queues and a list of capabilities. In this case, the record specifier uses a question mark ('?') indicating that the field is optional; if the record specifier uses a colon (':') then the field is required.

❸ Type batch_queue_list is an alias for a list of strings, but also contains a range limitation `[1..]`. This range limitation means that the list must contain at least one element.

❷ Type batch_capabilities is just an alias for a list of strings. It is a convenience type used to make the field description clearer.

❶ The template `declaration` uses the declaration modifier. This means that the template will only be executed once during the build of a particular machine profile. It also limits the content of the template to variable, function, and type definitions.

A complete set of types is now available for the batch configuration, but at this point, none of these types have been attached to a part of the configuration. The `bind` statement associates a particular type to a path. Note that a single path can have multiple type declarations associated with it. For the batch configuration, the `services/batch-server.pan` and `services/batch-worker.pan` have had a `bind` statement added.

```
template services/batch-server;

include { 'services/batch-types' };

bind '/batch/server' = batch_server;

'/batch/server/nodes/worker01.example.org/queues'
    = list('default');

'/batch/server/nodes/worker01.example.org/capabilities'
    = list('sw-license', 'fast-network');

'/batch/server/queues/default/maxCpuHours' = 1;
'/batch/server/queues/default/enabled' = true;
```

```
template services/batch-worker;

include { 'services/batch-types' };
```

```
bind '/batch/worker' = batch_worker;

'/batch/worker/server' = 'server.example.org';
'/batch/worker/enabled' = true;
```

Types have been bound to two paths with these `bind` statements. If any of the content does not conform to the specified types, then an error will occur during the compilation. Note that we have not limited the values for paths other than these two paths and their children. Configuration in other paths can be added without being subject to these type definitions. A global schema can be defined by binding a type definition to the root path '/'.

Default Values

Very often configuration parameters can have reasonable default values, avoiding the need to specify them explicitly within a machine profile. The pan type system allows default values to be defined and then inserted into a machine configuration when necessary. The following is a modified version of the `batch-types.pan` file with default values added.

```
declaration template services/batch-types;

type batch_capabilities = string[];

type batch_queue_list = string[1..];

type batch_node = {
  'queues' : batch_queue_list = list('default') ❶
  'capabilities' ? batch_capabilities
};

type batch_queue = {
  'maxCpuHours' : long(0..) = 1 ❷
  'enabled' : boolean = true ❸
};

type batch_server = {
  'nodes' : batch_node{}
  'queues' : batch_queue{} = nlist('default', nlist()) ❹
};

type batch_worker = {
  'server' : string
  'enabled' : boolean = true ❺
};
```

❶ If the queue list for a node is not specified, then assume that the node will participate in the 'default' queue. That is, the default value is a one-element list containing the string 'default'.

❷ Default to 1 CPU-hour for the queue execution limit.

❸ By default, a queue will be enabled.

❹ If no queues are specified, then provide an nlist containing only a queue definition for the 'default' queue. Note that the actual queue parameters are provided by the type definition batch_queue.

❺ By default, a worker will be enabled.

Using these default values, then simplifies the configuration templates `services/batch-server.pan` and `services/batch-worker.pan`.

```
template services/batch-server;

include { 'services/batch-types' };

bind '/batch/server' = batch_server;

'/batch/server/nodes/worker01.example.org/capabilities'
    = list('sw-license', 'fast-network');
```

```
template services/batch-worker;

include { 'services/batch-types' };

bind '/batch/worker' = batch_worker;

'/batch/worker/server' = 'server.example.org';
```

Compiling these templates will result in exactly the same generated files as with the previous configuration in which the default values were explicitly specified in the configuration. To use a value other than the default, the path just needs to be assigned the desired value. The defaults mechanism will never replace a value which was explicitly specified in the configuration.

Cross-Element and Cross-Machine Validation

Much of the power of using the pan language comes from its ability to ensure the consistency between different elements within a machine profile and between configurations of different machine profiles. In our example we have two cases where these types of validations would be useful: 1) the list of queues for a node should only reference defined queues and 2) the worker list on the server and the defined workers should be consistent.

The file `batch-types.pan` will be expanded to include validation functions for these cases. Each validation function must return `true` if the value is valid. If the value is not valid, then the function can return `false` or throw an exception via the `error` function. The `error` function allows you to provide a descriptive error message for the user. The contents of the modified file are:

```
declaration template services/batch-types;

function valid_batch_queue_list❶ = {
  foreach (index; queue_name; ARGV[0]) {
    if (!path_exists('/batch/server/queues/' + queue_name)) {
      return(false);
    };
  };
  true;
};

function valid_batch_node_nlist❷ = {
  foreach (hostname; properties; ARGV[0]) {
```

```
    path = 'profiles/' + hostname + ':/batch/worker';
    if (!path_exists(path)) {
      error(path + ' doesn''t exist');
      return(false);
    };
  };
  true;
};

function server_exists❸ = {
  return(path_exists('profiles/' + ARGV[0] + ':/batch/server'));
};

function server_knows_about_me❹ = {
  regex = '^profiles/(.*)$';
  if (match(OBJECT, regex)) {
    parts = matches(OBJECT, regex);
    path = 'profiles/' + ARGV[0] +
             ':/batch/server/nodes/' + parts[1];
    if (!path_exists(path)) {
      error(path + ' doesn''t exist');
    };
  } else {
    error(OBJECT + ' doesn''t match ' + regex);
  };
  true;
};

function valid_server❺ = {
  (server_exists(ARGV[0]) && server_knows_about_me(ARGV[0]));
};


type batch_capabilities = string[];

type batch_queue_list = string[1..];

type batch_node = {
  'queues' : batch_queue_list = list('default')
              with valid_batch_queue_list(SELF)❻
  'capabilities' ? batch_capabilities
};

type batch_queue = {
  'maxCpuHours' : long(0..) = 1
  'enabled' : boolean = true
};

type batch_server = {
  'nodes' : batch_node{} with valid_batch_node_nlist(SELF)❼
  'queues' : batch_queue{} = nlist('default', nlist())
};

type batch_worker = {
  'server' : string with valid_server(SELF)❽
  'enabled' : boolean = true
};
```

❶ The argument to this function is the batch queue list for a node. The function loops over the queue names and ensures that the associated path in the configuration exists. For example for the 'default' queue, the path '/batch/server/queues/default' must exist.

❷ The argument to this function is the nlist of worker nodes. The function loops over the worker node entries and constructs a path using the worker

node name. For example for the worker node 'worker01.example.org', it will construct the path 'worker01.example.org:/batch/worker'. This is an *external* path that references another machine profile. In this case, the server profile 'server.example.org' will reference all of the worker profiles, e.g. 'worker01.example.org'. If the node is configured as a worker, the path '/batch/worker' will exist on the node.

❸ The argument to this function is the name of the server as configured on a worker node. Similar to the previous function, this constructs a path on the referenced server and verifies that it exists. In this example, each worker will verify that the path 'server.example.org:/batch/server' exists.

❹ The argument to this function is also the name of the server as configured on a worker node. This function will extract the list of workers in the server configuration and ensure that the worker's name appears. This uses a regular expression to extract the machine name from the `OBJECT` variable, which contains the name of the object template being processed. The constructed path will exist if the server configuration contains the named worker node.

❺ The argument to this function is the name of the server. It is a convenience function that combines the previous two functions.

These functions are tied to a type definition using a `with` clause. The `with` clause will execute the given code block for the given type after the profile has been fully constructed. Usually, the code block will reference the special variable `SELF`, which contains the value associated with the given type. Although any block of code can be used in the type definition, it is best practice to define a validation function with the code and reference that validation function. This makes the type definition easier to read. The `with` clauses for the cross-element and cross-machine validation are:

❻ Run the `valid_batch_queue_list` function for all of the node queue lists.

❼ Run the `valid_batch_node_nlist` function for the server's node nlist.

❽ Run the `valid_server` function for the worker node's configured server.

This type of validation ensures internal and external consistency of machine configurations and can significantly enhance confidence in the defined configurations. Note that the cross-machine validation will work even with circular dependencies, allowing server and client validation for services.

Path Prefixes

Although in this particular example there is a limited number of parameters set, most real examples involve a large number of parameters and repetitive specifications of similar absolute paths. The `prefix` pseudo-statement is a convenience for reducing duplication in path specifications. The path provided in the `prefix` statement will be applied to any relative paths found in a template *after* the `prefix` statement.

As an example, we take the batch server configuration, adding a second worker node.

```
template services/batch-server;

include { 'services/batch-types' };

bind '/batch/server' = batch_server;

prefix '/batch/server/nodes';

'worker01.example.org/capabilities'
    = list('sw-license', 'fast-network');

'worker02.example.org/capabilities' = list();
```

In this case, this saves us from having to duplicate the prefix '/batch/server/nodes' for each worker node. Note that the prefix is expanded when the template is compiled and *does not* affect any included templates. Although multiple `prefix` statements can be used in a template, it is best practice to use only one near the beginning of the template.

CHAPTER 3

Core Syntax

As you will have seen in the whirlwind tour, a complete site or service configuration consists of a set of files called "templates". These files are usually managed via a versioning system to track changes and to permit reverting to an earlier state. The top-level syntax of the templates is especially simple: a template declaration followed by a list of statements that are executed in sequence. The compiler will serialize a machine profile, usually in XML format, for each "object" template it encounters.

Templates

Syntax

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

The syntax of a template file is simple:

```
[ modifier ] template template-name;
[ statement ... ]
```

where the optional modifier is either `object`, `structure`, `unique`, or `declaration`. There are five different types of templates that are identified by the template modifier; the four listed above and an "ordinary" template that has no modifier.

A template name is a series of substrings separated by slashes. Each substring may consist of letters, digits, underscores, hyphens, periods, and pluses. The substrings may not be empty or begin with a period; the template name may not begin or end with a slash.

Each template must reside in a separate file with the name *`template-name`*`.pan` with any terms separated with slashes corresponding to subdirectories. For example, a template with the name "service/batch/worker-23" must have a file name of `worker-23.pan` and reside in a subdirectory `service/batch/`.

Note

The older file extension "tpl" is also accepted by the pan compiler, but the "pan" extension is preferred. If files with both extensions exist for a given template, then the file with the "pan" extension will be used by the compiler.

Types of Templates

Object Templates

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

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

Ordinary Templates

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

Unique Templates

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

Declaration Templates

A template declared with a `declaration` modifier is a declaration template. These templates may contain only those pan statements that do not modify the machine profile. That is, they may contain only **type**, **bind**, **variable**, and **function** statements. A declaration template will only be executed once for each processed

object template no matter how many times it is included. It will be executed when the first include statement referencing the template is encountered.

Structure Templates

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

Comments

These files may contain comments that start with the hash sign ('#') and terminate with the next new line or end of file. Comments may occur anywhere in the file except in the middle of strings, where they will be taken to be part of the string itself.

Whitespace in the template files is ignored except when it is used to separate language tokens.

Statements

Assignment

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

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

where the path is represented by a string literal. Single-quoted strings are slightly more efficient, but double-quoted strings work as well.

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

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

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

A conditional form of the assignment statement also exists:

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

where the path is again represented by a string literal. The conditional form (?=) will only execute the DML block and assign a value if the named path does not exist or contains the `undef` value.

Prefix

The **prefix** (pseudo-)statement provides an absolute path used to resolve relative paths in assignment statements that occur afterwards in the template. It has the form:

```
prefix '/some/absolute/path';
```

The path must be an absolute path or an empty string. If the empty string is given, no prefix is used for subsequent assignment statements with relative paths. The **prefix** statement can be used multiple times within a given template.

This statement is evaluated at compile time and only affects assignment statements in the same file as the definition.

Include

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

```
include dml;
```

The DML block must evaluate to a string, `undef`, or `null`. If the result is `undef` or `null`, the **include** statement does nothing; if the result is a string, the named template is loaded and executed. Any other type will generate an error.

Ordinary templates may be included multiple times. Templates marked as `declaration` or `unique` templates will be only included once where first encountered. Includes which create cyclic dependencies are not permitted and will generate a fatal error.

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

Variable Definition

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

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

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

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

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

Any valid identifier may be used to name a global variable.

Caution

Global and local variables share a common namespace. Best practice dictates that global variables have names with all uppercase letters (e.g. `MY_GLOBAL_VAR`) and local variables have names with all lowercase letters (e.g. `my_local_var`). This avoids conflicts and unexpected errors when sharing configurations.

Function Definition

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

```
function identifier = dml;
```

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

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

Type Definition

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

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

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

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

Validation

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

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

See the Type section for a complete description of the *type-spec* syntax.

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

```
valid path = DML;
```

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

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

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

CHAPTER 4

Data Types

The data typing system forms the foundation of the validation features of the pan language. All configuration elements are implicitly typed based on values assigned to them. Types, once inferred, are enforced by the compiler.

Type Hierarchy

There are four primitive, atomic types in the pan language: boolean, long, double, and string. Additionally, there are three string-like types: path, link, and regular expression. These appear in special constructs and have additional validity constraints associated with them. All of these atomic types are known as "properties".

The language contains two types of collections: list and nlist. The 'list' is an ordered list of elements, which uses the index (an integer) as the key. The named list (nlist) associates a string key with a value; these are also known as hashes or associative lists. These collections are known as "resources".

The complete type hierarchy is shown in Figure 4.1, “Pan language type hierarchy”, including the two special types `undef` and `null`.

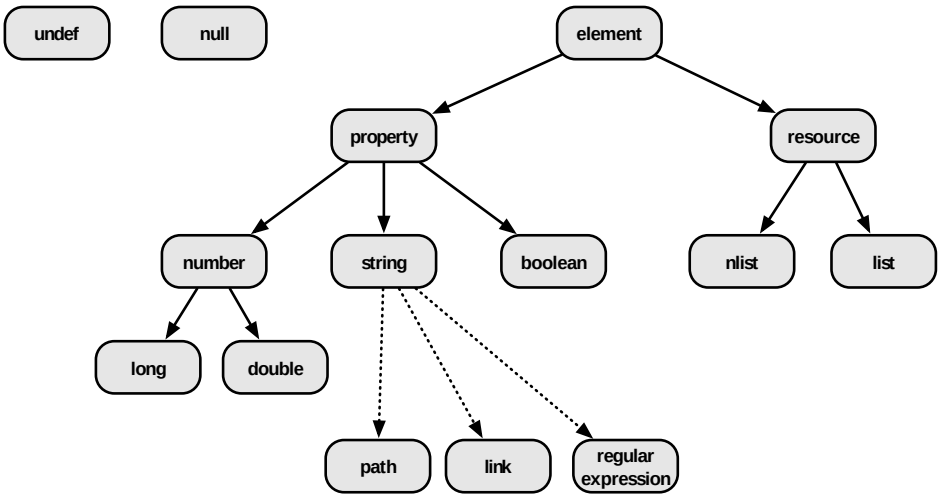

Figure 4.1. Pan language type hierarchy

Implicit Typing

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.

All of the elements in a configuration will have a concrete data type assigned to them. Usually this is inferred from the configuration itself. Once a concrete data type has been assigned to an element, the compiler will enforce the data type, disallowing replacement of a long value with a string, for instance. More detailed validation must be explicitly defined in the configuration (see the Validation chapter).

Properties and Primitive Types

Boolean Literals

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

Long Literals

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

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

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

Double Literals

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

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

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

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

String Literals

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

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

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

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

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

```
"foo"
"it's a sentence"
"Java-style escapes: \t (tab) \r (carriage return) \n (newline)"
"Java-style escapes: \b (backspace) \f (form feed)"
```

```
"Hexadecimal escapes: \x3d (=) \x00 (NULL byte) \x0A (newline)"
"Miscellaneous escapes: \" (double quote) \\ (backslash)"
"this string spans two lines and\
does not contain a newline"
```

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

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

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

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

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

```
'/system/binary/stuff' = base64_decode(<<EOT);
H4sIAOwLyDwAA02PQQ7DMAgE731FX9BT1f8Q
Z52iYhthEiW/r2SitCdmxCK0E3W8no+36n2G
8UbOrYYWGROCgurBe4JeCexI2ahgWF5rulaL
tImkDxbucS0tcc3t5GXMAqeZnIYo+TvAmsL8
GGLobbUUX7pT+pxkXJc/5Bx5p0ki7Cgq5Kcc
GrCR8PzruUfP2xfJgVqHCgEAAA==
EOT
```

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

String-Like Types

Path

Pan paths are represented as string literals; either of the standard quoted forms for a string literal can be used to represent a path. There are three different types of paths: external, absolute, and relative.

An *external path* explicitly references an object template. The syntax for an external path is:

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

where the substring before the colon is the template name and the substring after the colon is an absolute path. The leading slash of the absolute path is optional in an external path. This form will work for both namespaced and non-namespaced object templates.

An *absolute path* starts at the top of a configuration tree and identifies a node within the tree. All absolute paths start with a slash ("/") and are followed by a series of terms that identify a specific child of each resource. A bare slash ("/") refers to the

full configuration tree. The allowed syntax for each term in the path is described below.

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

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

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

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

Link

A property can hold a reference to another element; this is known as a link. The value of the link is the absolute path of the referenced element. A property explicitly declared to be a link will be validated to ensure that 1) it represents a valid absolute path and 2) that the given path exists in the final configuration.

Regular Expression

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

When the compiler can infer that a string literal must be a regular expression, it will validate the regular expression at compile time, failing when an invalid regular expression is provided.

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/0x2' = '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.

Special Types

The pan language contains two special types: `undef` and `null`.

The `undef` literal can be used to represent the undefined element, i.e. an element which is neither a property nor a resource. The undefined element cannot be written to a final machine profile and most built-in functions will report a fatal error when processing it. It can be used to mark an element that must be overwritten during the processing.

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

CHAPTER 5

Data Manipulation Language (DML)

Any non-trivial configuration will need to have some values that are calculated. The Data Manipulation Language (DML), a subset of the full pan configuration language, fulfills this role. This subset has the features of many imperative programming languages, but can *only* be used on the right-hand side of a statement, that is, to calculate a value.

DML Syntax

A DML block consists of one or more statements separated by semicolons. The block must be delimited by braces if there is more than one statement. The value of the block is the value of the last statement executed within the block. *All* DML statements return a value, even flow control statements like `if` and `foreach`.

Variables

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

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

```
# populate /table which is an 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 ('arouge')
  length(z);               # this will be 6
};
```

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

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

Caution

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

Operators

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

Table 5.1. Unary DML Operators

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

Table 5.2. Binary DML Operators

+	number	addition
+	string	string concatenation
-	number	subtraction
*	number	multiplication
/	number	division
%	long	modulus

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

Table 5.3. Operator Precedence (lowest to highest)

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

Flow Control

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

Caution

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

Branching (`if` statement)

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

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

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

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

Looping (`while` and `for` statements)

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

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

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

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

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

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

Note that the compiler enforces an iteration limit to avoid infinite loops. Loops exceeding the iteration limit will cause the compiler to abort the execution. The value of this limit can be set via a compiler option.

Iteration (`foreach` statement)

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

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

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

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

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

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

CHAPTER 6

Functions

The pan configuration has a rich set of built-in functions for manipulating elements and for debugging. In addition, user-defined functions can be specified, which are often used to make configurations more modular and maintainable.

Built-In Functions

Built-in functions are actually treated as operators within the DML language. Because of this, they are highly optimized and often process their arguments specially. In all cases, users should prefer built-in functions to user-defined functions when possible. The following tables describe all of the built-in functions; refer to the appendix to see the arguments and other detailed information about the functions.

Table 6.1. String Manipulation Functions

file_contents(3)	Lookup the named file and provide the file's contents as a string.
format(3)	Generate a formatted string based on the formatting parameters and the values provided.
index(3)	Return the index of a substring or -1 if the substring is not found.
length(3)	Gives the length of a string.
match(3)	Return a boolean indicating if a string matches the given regular expression.
matches(3)	Return an array containing the matched string and matched groups for a given string and regular expression.
replace(3)	Replace all occurrences of a substring within a given string.
splice(3)	Remove a substring and optionally replace it with another.
split(3)	Split a string based on a given regular expression and return an array of the results.

substr(3)	Extract a substring from the given string.
to_lowercase(3)	Change all of the characters in a string to lowercase (using the US locale).
to_uppercase(3)	Change all of the characters in a string to uppercase (using the US locale).

Table 6.2. Debugging Functions

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

Table 6.3. Encoding and Decoding Functions

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

Table 6.4. Resource Manipulation Functions

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

next(3)	Extract the next value while iterating over a resource. Returns a boolean to indicate if more values exist in the resource.
prepend(3)	Add a value to the beginning of a list.
splice(3)	Remove a section of a list and optionally replace removed values with those in a given list.

Table 6.5. Type Checking Functions

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

is_string(3)	Check if the argument is a string value. If the argument is a simple variable reference and the referenced variable does not exist, the function will return false rather than raising an error.

Table 6.6. Type Conversion Functions

to_boolean(3)	Convert the argument to a boolean. Any number other than 0 and 0.0 is `true`. The empty string and the string 'false' (ignoring case) return `false`. Any other string will return `true`. If the argument is a resource, an error will occur.
to_double(3)	Convert the argument to a double value. Strings will be parsed to create a double value; any literal form of a double is valid. Boolean values will convert to `0.0` and `1.0` for `false` and `true`, respectively. Long values are converted to the corresponding double value. Double values are unchanged.
to_long(3)	Convert the argument to a long value. Strings will be parsed to create a long value; any literal form of a long is valid (e.g. hex or octal literals). Boolean values will convert to `0` and `1` for `false` and `true`, respectively. Double values are rounded to the nearest long value. Long values are unchanged.
to_string(3)	Convert the argument to a string. The function will return a string representation for any argument, including list and nlist.

Table 6.7. Miscellaneous Functions

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

User-Defined Functions

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

```
function identifier = DML;
```

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

When the function is called, the DML will have the variables ARGC and ARGV defined. The variable ARGC contains the number of arguments passed to the function; ARGV is a list containing the values of the arguments.

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

```
f(null);
```

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

The pan language does *not* check the number or types of arguments automatically. The DML block that defines the function must make all of these checks explicitly and use the error function to emit an informative message in case of an error.

Recursive calls to a function are permitted. However, the call depth is limited (by an option when the compiler is invoked) to avoid infinite recursion. Typically, the maximum is a small number like 10. Recursion is expensive within the pan language and should be avoided if possible.

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

```
function paired_numbers = {

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

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

  'ok';

};
```

CHAPTER 7

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.

Forcing Validation

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.

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

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

where *path* is a valid path name and *type-spec* is either a type specification or name of an existing type.

Full type specifications are of the form:

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

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

Implicit Typing

If you worked through the previous chapters, 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. The compiler will infer an element's data type from the first value assigned to it. From then on it will enforce that type, raising an error if, for instance, a double is replaced by a string. If necessary, the implicit type can be removed from an element by assigning it to `undef` before changing the value.

Binding Primitive Types to Paths

Downstream machine configuration tools will likely expect parameters to have certain types, producing errors or erroneous configurations if the correct type is not used. One of the strengths of the pan language is to specify explicit constraints on the element to detect problems before configurations are deployed to machines.

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 short-hand for `3..3`). A range can be applied to a long, double, or string type definition. For strings, the range is applied to the length of the string.

User-Defined Types

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

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

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

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

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

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

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

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

Slightly more complex is to create uniform collections:

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

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

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

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

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

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

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

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.

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`, which was defined in the next section.

Validation Functions

To simplify type definitions, validation functions are often defined. These are user-defined functions defined using the standard **function** statement. They can be referenced within a type definition just as they would be in any DML block. However, validation functions *must* return a boolean value or raise an error with the `error` function. A validation function that returns a non-boolean value will abort the compilation. Similarly, a validation function that returns `false` will raise an error indicating that the value for the tested element is invalid.

A validation function that checks that a value is a valid IPv4 address 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.

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.

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;
};
```

```
# 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.

Schemas

The pan language allows complete configuration schema to be defined. Actually, you are capable of doing this already as defining a schema is nothing more than defining a type and binding that type to the root element. An example of this is:

```
object template schema_example;

include { 'type_definitions' };

type schema = {
  'software' : software_type
  'hardware' : hardware_type
  'packages' : packages_type
};

bind '/' = schema;

# Actual definitions of parameters.
# ...
```

In this fictitious example, the concrete types would be defined in the included file and the template would actually define the configuration parameters.

CHAPTER 8

Modular Configurations

Defining the configuration for a machine with many services, let alone a full site, quickly involves a large number of parameters. Often subsets of the configuration can be shared between services or machines. To minimize duplication and encourage sharing of configurations, the pan language has features to allow modularization of the configuration.

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, variable, and function declarations. 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.pan`; 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 here, 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.

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.

CHAPTER 9

Advanced Features

This chapter discusses annotations and logging, two advanced topics that can be used to facilitate the management of sites and better understand a site's configuration.

Annotations

The compiler supports pan language annotations and provides a mechanism for recovering those annotations in a separate XML file. While the compiler permits annotations to occur in nearly any location in a source file, only annotations attached to certain syntactic elements can be recovered. Currently these are those before the template declaration, variable declarations, function declarations, type declarations, and field specifications. Examples of all are in the example file.

```
@maintainer{
  name = Jane Manager
  email = jane.manager@example.org
}
@{
  Example template that shows off the
  annotation features of the compiler.
}
object template annotations;

@use{
  type = long
  default = 1
  note = negative values raise an exception
}
variable VALUE ?= 1;

@documentation{
  desc = simple addition of two numbers
  arg = first number to add
  arg = second number to add
}
function ADD = {
 ARGV[0] + ARGV[1];
};

type EXTERN = {
  'info' ? string
```

```
};

@documentation{
  Simple definition of a key value pair.
}
type KV_PAIR = extensible {

  @{additional information fields}
  include EXTERN

  @{key for pair as string}
  'key' : string

  @{value for pair as string}
  'value' : string = to_string(2 + 3)
};

bind '/pair' = KV_PAIR;

'/add' = ADD(1, 2);

'/pair/key' = 'KEY';
'/pair/value' = 'VALUE';
```

The command will produce one output file for each source file, using the directory hierarchy of the source files, *not the namespace hierarchy*. When processing the files, you must provide both the desired output directory (which must exist) using the `--annotation-dir` option, as well as the root file system directory for all of the processed files with the `--annotation-base-dir` option. Using the following command to process the file:

```
$ panc \
  --annotation-dir=annotations \
  --annotation-base-dir=. \
  annotations.pan
```

will produce the following output in the file `annotations.pan.annotation.xml` (with whitespace and indentation added for clarity).

```
<?xml version="1.0" encoding="UTF-8"?>
<template xmlns="http://quattor.org/pan/annotations"
          name="annotations"
          type="OBJECT">
    <desc>
  Example template that shows off the
  annotation features of the compiler.
</desc>

    <maintainer>
        <name>Jane Manager</name>
        <email>jane.manager@example.org</email>
    </maintainer>

    <variable name="VALUE">
        <use>
            <type>long</type>
            <default>1</default>
            <note>negative values raise an exception</note>
        </use>
    </variable>

    <function name="ADD">
        <documentation>
```

```
            <desc>simple addition of two numbers</desc>
            <arg>first number to add</arg>
            <arg>second number to add</arg>
        </documentation>
    </function>

    <type name="EXTERN">
        <basetype extensible="no">
            <field name="info" required="no">
                <basetype name="string" extensible="no"/>
            </field>
        </basetype>
    </type>

    <type name="KV_PAIR">
        <documentation>
            <desc>
  Simple definition of a key value pair.
</desc>
        </documentation>

        <basetype extensible="yes">
            <include name="EXTERN"/>
            <field name="key" required="yes">
                <desc>key for pair as string</desc>
                <basetype name="string" extensible="no"/>
            </field>
            <field name="value" required="yes">
                <desc>value for pair as string</desc>
                <basetype name="string" extensible="no"/>
            </field>
        </basetype>

    </type>
    <basetype name="KV_PAIR" extensible="no"/>
</template>
```

The output filename includes the full input filename because variants with different suffixes may be present.

Logging

It is possible to log various activities of the pan compiler. The types of logging that can be specified are:

task — Task logging can be used to extract information about how long the various processing phases last for a particular object template. The build phases one will see in the log file are: execute, defaults, valid1, valid2, xml, and dep. There is also a build stage that combines the execute and defaults stages.

call — Call logging allows the full inclusion graph to be reconstructed, including function calls. Each include is logged even if the include would not actually include a file because the included file is a declaration or unique template that has already been included.

include — Include logging only logs the inclusion of templates and does not log function calls.

memory Memory logging show the memory usage during template processing. This can be used to see the progression of memory utilization and can be correlated with other activities if other types of logging are enabled.

all Turns all types of logging on.

none Turns all types of logging off.

Note that a log file name must also be specified, otherwise the logging information will not be saved.

The logging information can be used to understand the performance of the compiler and find bottlenecks in the configuration. It can also be used to extract information about the relationships between templates, which are then commonly passed to visualization tasks to allow a better understanding of the configuration. Many examples are included in the distribution as analysis scripts. See the command reference appendix for details.

Build Metadata

It is sometimes useful to be able to inject values into the compiled profiles without having to explicitly include a template into each object template. This is particularly appropriate for metadata like build numbers, build times, build machines, etc. This can be achieved by setting the root element that is used to start the build of all profiles. Use the `rootElement` attribute for ant and the `--root-element` option for the command line. The value must be a DML expression that evaluates to an nlist. For example, this expression

```
nlist('build-metadata', nlist('number', 1, 'date', '2012-01-01'))
```

would result in having the paths `/build-metadata/number`, `/build-metadata/date` being set to `1` and `2012-01-01`, respectively, in all object templates.

Caution

Values inserted into the profiles in this way are still subject to the usual validation. When inserting values, they must obey the schema you have defined for the profile.

CHAPTER 10

Performance Considerations

As configurations become larger, the speed at which the full configuration can be compiled becomes important. The logging features presented in the previous chapter can help identify slow parts of the compilation for you particular configuration. This chapter contains general advice on making the compilation as quick as possible.

Use Specific Paths

Whenever possible, use the most specific path and assign a property to that path. The code:

```
'/path' = nlist('a', 1, 'b', 2);
```

and the block:

```
'/path/a' = 1;
'/path/b' = 2;
```

provide identical results, although the second example is easier to read and will be better optimized by the compiler.

Use Escaped Literal Path Syntax

In previous versions of the compiler, it was necessary to use a DML block when part of a path needed to be escaped:

```
'/path' = nlist(escape('a/b'), 1);
```

Newer versions of the compiler provide a literal path syntax in which escaped portions can be written explicitly:

```
'/path/{a/b}' = 1;
```

This is both more legible and faster.

Use Built-In Functions

Built-in functions are significantly faster than equivalents defined with the pan language. In particular, the functions `append` and `prepend` should be used for incrementally building up lists (in preference to `push` equivalents). There are also functions like `to_uppercase` and `to_lowercase` that avoid character by character manipulation of strings.

The list of available built-in functions continues to expand. Check the list of functions with each new release of the compiler.

Invoking the Compiler

There are several ways to invoke the compiler, either from the command line, from ant, or from maven. For single, infrequent invocations of the compiler they are roughly equivalent in startup time. However, if the compiler will be invoked frequently it is better to avoid using the command line panc script. The reason for this is that the panc script starts a new JVM each time it is invoked, while the ant and maven invocations can reuse their own JVM. This means that for the panc script, you will pay the startup costs each time it is invoked while for ant or maven you pay it them only once. The startup costs are particularly expensive if you request a large amount of memory and do hundreds of compilations at a time.

Avoid Copying SELF

Assignments of `SELF` to a local variable inside of a code block will cause a deep copy of `SELF`. In the following code, the local variable `copy` will contain a complete replica of `SELF`.

```
'/path' = {
 copy = SELF;
 copy;
};
```

These copies can be time-consuming when `SELF` is a large resource or when the code is executed frequently. If you manipulate `SELF` within a code block, *always* reference `SELF` directly.

Also be aware that `copy` and `SELF` will contain independent copies so that changes to `copy` to not affect `SELF` and vice versa. This can lead to bugs that are difficult to find.

CHAPTER 11

Common Idioms

As you use the pan configuration, you will discover certain idioms which appear. This chapter describes some of the common idioms so that you can take advantage of them from the start and not need to rediscover them yourself.

Configuration File Templates

Although it is much better to create an abstracted schema for service configuration, practically it is often useful to directly embed a configuration file directly in the service configuration. In previous versions of the compiler, the configuration file was often created incrementally in a global variable and then assigned to a path. Something like the following was common:

```
variable USER = 'smith';
variable QUOTA = 10;

variable CONTENTS = <<EOF;
alpha = 1
beta = 2
EOF

variable CONTENTS = CONTENTS +
    'user = ' + USER + "\n";

variable CONTENTS = CONTENTS +
    'quota = ' + to_string(QUOTA) + "\n";


'/cfgfile' = CONTENTS;
```

This can be improved somewhat by using the `format` function:

```
variable USER = 'smith';
variable QUOTA = 10;


variable CFG_TEMPLATE = <<EOF;
alpha = 1
beta = 2
user = %s
quota = %d
```

```
EOF

'/cfgfile' = format(CFG_TEMPLATE, USER, QUOTA);
```

This can be further improved by moving the configuration template completely out of the pan language file. For instance, create the file `cfg-template.txt`:

```
alpha = 1
beta = 2
user = %s
quota = %d
```

which can then be used like this:

```
variable USER = 'smith';
variable QUOTA = 10;

'/cfgfile' = format(file_contents('cfg-template.txt'),
                    USER, QUOTA);
```

This is much easier to read and to maintain. It is especially helpful when the included configuration file has a syntax for which an external editor can provide additional help with validation.

Extension Templates

Often sets of templates that are intended for reuse will allow the configuration to be extended or modified at particular points by including named templates. For example, the following provides pre-configuration and post-configuration service hooks:

```
template my_service/config;

include if_exists('my_service/prehook');

# bulk of real service configuration

include if_exists('my_service/posthook');
```

In both of these cases, the named templates will be included if they can be found on the loadpath. If they are not found, the includes do nothing.

Global Variables as Switches

Configuration intended for reuse also tends to expose switches for common configuration options. The idiom looks like the following:

```
template my_service/config;

variable MY_OPTION ?= false;

'/my_service/config/my_option' =
  if (MY_OPTION) {
    'some value';
  } else {
    'some other value';
```

```
  };
};
```

In cases where the path simply should not exist if the option is not set, then using a default value of null can be the best option:

```
template my_service/config;

variable MY_OPTION ?= null;

'/my_service/config/my_option' = MY_OPTION;
```

In this case, if the variable MY_OPTION is not set to a value before executing this template, the null value will be used and the given path will simply be deleted.

Tri-state Variables

Occasionally is is useful to have tri-state variables. The most convenient values to use in this case are `true`, `false`, and `null`. With these values as the three states, you can use `is_null` to test explicitly for the third state. Using `undef` for the third value can cause problems because variables are automatically set to `undef` before executing a variable assignment statement.

CHAPTER 12

Troubleshooting

Compilation Problems

In a production environment, the number of templates and their complexity will be must greater. Often something goes wrong with the compilation or build resulting in one or more errors appearing on the console (standard error stream). There are four categories of errors:

Syntax Error	These include any errors that can be caught during the compilation of a single template. These include lexing, parsing, and syntax errors, but also semantic errors like absolute assignment statements appearing in a structure template that can be caught at compilation time.
Evaluation Error	These are the most common; these include any error that happens during the "execution" phase of processing like mathematical errors, primitive type conflicts, and the like. Usually the name of the template and the location where the error occurred will be included in the error message.
Validation Error	Validation errors occur during the "validation" phase and indicate that the generated machine profile violates the defined schema. Information about what type specification was violated and the offending path will be included in the error message.
System Error	These include low-level problems like problems reading from or writing to the file system.

In general, the errors try to indicate as precisely as possible the problem. Usually the name of the source file as well as the location inside the file (line and

column numbers) are indicated. For most evaluation exceptions, a traceback is also provided. Validation errors are the most terse, giving only the element causing the problem and the location of the type definition that has been violated.

There is one further class of errors called "compiler errors". These indicate an error in the logic of the compiler itself and should be accompanied by a detailed error message and a Java traceback. All compiler errors should be reported as a bug. The bug report should include the template that caused the problem along with the full Java traceback. Hopefully, you will not encounter these errors.

Common Problems

1.1. "Java Heap Space" warnings appear on console.

If you see messages that refer to "Java Heap Space" while running the compiler, then the java virtual machine does not have enough memory to compile the given templates. You must increase the amount of memory allocated to the java virtual machine when you start the compiler. See the section Running the Compiler for how to specify the VM memory.

1.2. The compilation is extremely slow.

If the compilation appears to be slow, check that the compiler is not thrashing because of a limited amount of memory. With the verbose option set, successful compilations will produce a summary like:

```
2 templates
2/2 compiled, 2/2 xml, 0/0 dep
0 errors, 166 ms, 0 MB/63 MB heap, 12 MB/116 MB nonheap
```

The last line with gives the maximum amount of heap memory used and the maximum available (the value marked "heap"). If the maximum used is more than about 80% of the maximum available, then you should consider increasing the memory allocated to the java virtual machine. See the section Running the Compiler for how to specify the VM memory.

1.3. "missing modifyThread Permission" warnings appear on console.

The java-implementation of the pan language compiler is completely multi-threaded. Internally, it controls several thread pools to handle compilation, execution, and serialization in parallel. At the end of a compilation, the compiler will normally destroy the thread pools that were created. The java security model requires that a program have the "modifyThread" permission to destroy threads. In some environments (notably Eclipse), this permission may not be given to the compiler. If this is the case, then the message "WARNING: missing modifyThread permission" is printed on the standard error. Lacking this permission causes a "thread leak", but the effects are

minor unless an extremely large number of templates are being compiled. If this is the case, then you should either change the configuration to grant this permission to the compiler, or work in an environment that grants it by default (e.g. using ant from the command line).

This problem is fixed if using Java6. If you have several JREs installed, be sure to configure Eclipse to use Java 6. Go to Window → Preferences → Java → Installed JREs. If you don't see the JRE you want (and you have it installed), use the "Search" button to have eclipse configure the new JRE for you. Make sure you select it after it is found.

1.4. Unnecessary rebuild of clusters

It can happen that a cluster is always rebuilt when you run ant, even if there was no change in the dependencies. In this case, you may suspect a Java issue with optimizations enabled by default (JIT). The only workaround is to disable these optimizations by adding the option -Xint to Java VM when running ant. It is achieved differently depending how you started ant:

- From command line: define environment variable ANT_OPTS.
- From Eclipse: right click on build.xml in ant pane, choose Run As... → External Tools... and then click on JRE tab. Be sure to use a separate JRE (if possible Java 6 or later) and add option in the options area.

This problem has been seen on Windows only, with Java 5 and Java 6.

Bug Reporting

The pan compiler, like all software, contains bugs. If the problem your experiencing looks to be misbehavior by the compiler, please report the problem. Bug reports can be filed in the standard Quattor bug tracking system on SourceForge. When submitting a bug, please use the following options to be sure that the bug is noticed as soon as possible.

```
Project: quattor
Category: panc
Assigned to: loomisc
```

to ensure that the bug is treated as soon as possible. Bug fixes are generally rolled into the next planned release. Major releases are scheduled every six months. Bug fix releases are planned monthly, if needed.

APPENDIX A

Obtaining the Compiler

Binary Distributions

Binary packages for all releases are available from SourceForge in a variety of formats:

```
http://sourceforge.net/projects/quattor/files/
```

The same location also contains documentation for the compiler. This document is also bundled in the distribution files.

Source

The source for the pan compiler is managed through a git repository. The software can be checked out with the following command:

```
git clone git://quattor.git.sourceforge.net/gitroot/quattor/quattor/pan
```

This provides a *read-only* copy of the pan repository. If you need write access to the repository, consult the SourceForge documentation to find how to checkout a hosted git repository with write access. You will need to be a member of the Quattor SourceForge project.

The master branch is the main development branch. Although an effort is made to ensure that this code functions correctly, there may be times when it is broken. Released versions can be found through the named branches and tags. Use the git commands:

```
git branch -r
git tag -l
```

to see the available branches and tags.

Building

Correctly building the Java-implementation of the pan compiler requires version 1.5.0 or later of a Java Development Kit (JDK). Many linux distributions include the GNU implementation of Java. *The GNU implementation cannot build or run the pan compiler correctly.* Full versions of Java for linux, Solaris, and Windows can be obtained from Oracle. Maven can be obtained from the Apache Foundation web site.

The build of the compiler is done via Apache Maven that also depends on Java. For Maven to find the correct version of the compiler, the environment variable `JAVA_HOME` should be defined:

```
export JAVA_HOME=<path to java area>
```

or

```
setenv JAVA_HOME <path to java area>
```

depending on the type of shell that you use. After that, the entire build can be accomplished with:

```
mvn clean package
```

where the current working directory is the root of the directory checked out from subversion. The default build will compile all of the java sources, run the unit tests, and package the compiler. Tarballs (plain, gzipped, and bzipped) as well as a zip file are created on all platforms. The build will also create an RPM on platforms that support it. The final packages can be found in the `target` subdirectory.

> **Note**
>
> Current builds of the compiler are done with Maven 2.2.1. The builds have not yet been tested with the Maven 3 releases.

Installation

The proper installation of the pan compiler depends on how it will be used. If it will be used from the command line (either directly or through another program), then the full installation from a binary package should be done. However, if the compiler will be run via **ant**, then one really only needs to install the `panc.jar` file.

Full Package Installation

Once you have a binary distribution of the compiler (either building it from source or downloading a pre-built version), installation of the java compiler

should be relatively painless. The binary packages include the code, scripts, and documentation of the compiler.

Tarballs/Zip File. Untar/unzip the package in a convenient area and redefine the `PATH` variable to include the `bin` subdirectory. You should then have access to **panc** and the various log file analysis scripts from the command line.

RPM. Simply using the command **rpm** (as root) to install the package will be enough. The scripts and binaries will be installed in the standard locations on the system. The RPM is not relocatable. If you need to install the compiler as a regular user, use one of the machine-independent packages.

Using the compiler requires Java 1.5.0 or later to be installed on the system. If you want to run the compiler from ant, then you must have ant version 1.7.0 or later installed on your system.

Eclipse Integration

To integrate the compiler in an Integrated Development Environment (IDE) like eclipse, only the file `panc.jar` is needed, presuming that the compiler will be called via the ant task. Build files that reference the compiler must define the panc task and then may use the task to invoke the compiler. See the documentation for invoking the compiler from ant.

APPENDIX B

Running the Compiler

The performance of the compiler can degrade markedly if there is not sufficient memory to do a particular compile and build. Moreover, the default memory allocation and vary wildly depending on how and when the compiler is invoked. Similarly, there are other options that may improve the performance of the compiler. For instance, it is usually advisable to use the `-server` option.

Command Line

The compiler can be invoked from the command line by using **panc**. This is a script that is installed with the pan compiler package that invokes a Java virtual machine and the compiler. The script options have been designed to be as compatible with previous versions of the **panc** command as possible.

The full list of options can be obtained with the `--help` option or by looking on the relevant man page.

Using java Command

If the Java compiler class is being directly invoked via the **java** command, then the option `-Xmx` must be used to change the VM memory available (for any reasonably sized compilation). For example to start **java** with 1024 MB of memory, the following command and options can be used:

```
java -Xmx1024M org.quattor.pan.Compiler [options...]
```

The same can be done for other options.

Maven

The pan compiler release now contains a simple maven plug-in that will perform a pan syntax check and build on a simple set of files. The plug-in is available from

the central maven repository. To use this, you will need to configure maven for that repository. A maven archetype is also provided that can be used to generate a working skeleton that demonstrates the pan maven plugin.

To generate a skeleton maven project from the archetype use the following command (use the latest version of the archetype):

```
$ mvn archetype:generate \
  -DarchetypeArtifactId=panc-maven-archetype \
  -DarchetypeGroupId=org.quattor.pan \
  -DarchetypeVersion=9.2

...

Define value for property 'groupId': : org.example.pan
Define value for property 'artifactId': : mysite
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  org.example.pan: :
Confirm properties configuration:
groupId: org.example.pan
artifactId: mysite
version: 1.0-SNAPSHOT
package: org.example.pan
 Y: :

...

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 19.690s
[INFO] Finished at: Mon Feb 20 08:23:52 CET 2012
[INFO] Final Memory: 9M/81M
[INFO] ------------------------------------------------------------------------
```

As can be seen above, the process will ask for general information about the project that you want to create. The process should end with a "BUILD SUCCESS" and create a subdirectory with the maven project. In the example, the subdirectory (and artifactId) are named "mysite".

Within this subdirectory ("mysite"), you can then invoke the entire build process by doing the following:

```
$ cd mysite/
$ mvn clean install

...

[INFO] --- panc-maven-plugin:9.2-SNAPSHOT:pan-check-syntax (check-syntax) @ mysite ---
[INFO]
[INFO] --- panc-maven-plugin:9.2-SNAPSHOT:pan-build (build) @ mysite ---

...

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.782s
[INFO] Finished at: Mon Feb 20 08:27:51 CET 2012
[INFO] Final Memory: 3M/81M
[INFO] ------------------------------------------------------------------------
```

Again, this should end with a "BUILD SUCCESS". It will have generated the machine profile in the `target/profiles/node.example.org.xml` file:

```
$ cat target/profiles/node.example.org.xml

<?xml version="1.0" encoding="UTF-8"?>
<nlist format="pan" name="profile">
    <list name="alpha">
        <long>1</long>
        <long>2</long>
        <long>3</long>
        <long>4</long>
    </list>
    <nlist name="beta">
        <string name="delta">OK</string>
        <boolean name="epsilon">true</boolean>
        <string name="gamma">OK</string>
        <double name="zeta">3.14</double>
    </nlist>
</nlist>
```

The `pom.xml` file in the skeleton provides a good example on how to run the plugin. You can also obtain more detailed help via the maven help system:

```
$ mvn help:describe -Dplugin=panc -Ddetail=true
```

Ant

Using an ant task to invoke the compiler allows the compiler to be easily integrated with other machine management tasks. To use the pan compiler within an ant build file, the pan compiler tasks must be defined. This can be done with a tasks like:

```
<target name="define.panc.task">

  <taskdef resource="org/quattor/ant/panc-ant.xml">
    <classpath>
      <pathelement path="${panc.jar}" />
    </classpath>
  </taskdef>

</target>
```

where the property ${panc.jar} points to the jar file `panc.jar` distributed with the pan compiler release.

There are two tasks defined: **panc** and **panc-check-syntax**. The first provides all of the functionality available through the compiler with a large number of options. The second focuses on testing the pan language syntax and takes a very limited number of options. Running the compiler can be done with tasks like the following:

```
<target name="compile.cluster.profiles">

  <!-- Define the load path.  By default this is just the cluster area. -->
  <path id="pan.loadpath">
    <dirset dir="${basedir}" includes="**/*" />
  </path>

  <panc-check-syntax ...options... >
    <fileset dir="${basedir}/profiles" casesensitive="yes" includes="*.pan" />
  </panc-check-syntax>
```

```
  <panc ...options... >
    <path refid="pan.loadpath" />
    <fileset dir="${basedir}/profiles" casesensitive="yes" includes="*.pan" />
  </panc>

</target>
```

where `...options...` is replaced with valid options for the pan compiler ant tasks.

The following table summarizes the ant task attributes, only the `deprecationLevel`, `failOnWarn` and `verbose` options are permitted for the **panc-check-syntax** task.

Table B.1. Ant Task Attributes

callDepthLimit	Maximum number of recursive calls.	No. Default value: 5000
checkDependencies	Whether or not to check dependencies and only build profiles that have not changed.	No. Default value: true
debugTask	Emit debugging messages for the ant task itself. If the value is 1, then normal debugging is turned on; if the value is greater than 1 then verbose debugging is turned on. A value of zero turns off the task debugging.	No. Default value: 0
deprecationLevel	Set deprecation level. A negative number turns off deprecation warnings. 0 prints warnings for deprecated features for next release, 1 for second release, etc.	No. Default value: 0
failOnWarn	If set to true, then warnings will be treated like errors and abort the compilation.	No. Default value: false
depWriteEnabled	Whether or not to write dependency file.	No. Default value: false
dumpAnnotations	Whether or not to write annotation information to the standard output.	No. Default value: false

forceBuild	Force the build of all given object templates.	No. Default value: false
formatter	The formatter to use for the output file. The accepted values are: "pan", "json", "text", and "dot". The value "xmldb" is accepted, but deprecated.	No. Default value: pan
gzipOutput	Whether or not to gzip the (XML) output file.	No. Default value: no
ignoreDependencyPattern	A pattern which will select dependencies to ignore during the task's dependency calculation. The pattern will be matched against the namespaced template name.	No. Default value: null
includeRoot	Directory to use as the root of the compilation.	Yes.
includes	Set of directories below the include root to use in the compilation. This is a "glob".	Yes.
iterationLimit	Set the maximum number of iterations. This is a failsafe to avoid infinite loops.	No. Default value: 5000
logfile	The name of the file to use for logging information. This value must be defined in order to enable logging.	Yes, if logging attribute is used.
logging	Enable different types of logging. The possible values are: "all", "none", "include", "call", "task", and "memory". Multiple values may be included as a comma-separated list. The value "none" will override any other setting.	No.

nthread	Number of threads to use while compiling. Use 0 to set the number of threads to the number of CPUs/ cores on the machine.	No. Default value: 0
outputDirectory	The directory that will contain the output of the compilation.	Yes.
sessionDirectory	Set the session directory for the build.	No.
verbose	Whether to include a summary of the compilation, including number of profiles compiled and overall memory utilization.	No. Default value: false
xmlWriteEnabled	Whether or not to write the output files from the compilation.	No. Default value: true
dumpAnnotations	Write the annotations to the standard output as the template is compiled.	No. Default value: false
annotationDirectory	Directory which will hold files containing annotation information.	No.
batchSize	If set to a positive integer, the outdated templates will be processed in batches of batchSize.	No. Default value: 0
rootElement	A compile-time expression that evaluates to an nlist. This nlist is used as the root nlist for all compiled object templates. A convenient mechanism for injecting build numbers and other metadata into the profiles.	No. Default value: null (empty nlist)

Nested Elements

Some of the configuration options are specified via nested elements. The **panc** task supports all of these; the **panc-check-syntax** task only supports the `fileset` nested element.

Debug Element

The debug element is used to enable debugging output from the `debug` and `traceback` functions. The element takes the `include` and `exclude` attributes, both of which are optional. The `debug` element may appear multiple times within the task with the final list of `include` and `exclude` patterns being the union of all of those specified in the elements. The values for the attributes are regular expressions. For example,

```
<debug include="xen/.*" exclude=".*unwanted.*" />
```

will cause `debug` and `traceback` functions in the xen namespace to emit messages as long as the template name does not contain the string "unwanted". A special case is,

```
<debug>
```

that behaves exactly like:

```
<debug include=".*" exclude="^$" />
```

That is, it will turn on debugging in all templates, unless excluded via another `debug` element.

Fileset

Nested `fileset` elements specify the list of files to process with the compiler. These are standard ant element and take all of the usual attributes.

Path

A nested `path` element specifies the list of include directories to use during the compilation. This is a standard ant element and takes all of the usual attributes.

Setting JVM Parameters

If the compiler is invoked via the pan compiler ant task, then the memory option can be added with the `ANT_OPTS` environmental variable.

```
export ANT_OPTS="-Xmx1024M"
```

or

```
setenv ANT_OPTS "-Xmx1024M"
```

depending on whether you use a c-shell or a bourne shell. Other options can be similarly added to the environmental variable. (The value is a space-separated list.)

Invocation Inside Eclipse

If you use the default VM to run the pan compiler ant task, then you will need to increase the memory when starting eclipse. From the command line you can add the VM arguments like:

```
eclipse -vmargs -Xmx<memory size>
```

You may also need to increase the memory in the "permanent" generation for a Sun VM with

```
eclipse -vmargs -XX:MaxPermSize=<memory size>
```

This will increase the memory available to eclipse and to all tasks using the default virtual machine. For Max OS X, you will have to edit the application "ini" file. See the eclipse instructions for how to do this.

If you invoke a new Java virtual machine for each build, then you can change the ant arguments via the run parameters. From within the "ant" view, right-click on the appropriate ant build file, and then select "Run As -> Ant Build...". In the pop-up window, select the JRE tab. In the "VM arguments" panel, add the `-Xmx` option. The next build will use these options. Other VM options can be changed in the same way.

The options can also be set using the "Window -> Preferences -> Java -> Installed JREs" panel. Select the JRE you want use, click edit and add the additional parameters in the "DefaultVM arguments" field.

APPENDIX C

Command Reference

The pan distributions provide a set of commands that allow the compiler to be invoked and that demonstrate how to analyze available logging information. These commands are provided for ease of use for one-off tasks. The compiler can be more efficiently invoked via Apache Ant or Maven for automated use of the compiler in production.

Name

panc — compile pan language templates

Synopsis

`panc` [-d | --debug] [--debug-include *`regex`*] [--debug-exclude *`regex`*] [--root-element *`nlist-dml`*] [--dump-annotations] [--annotation-dir *`dir`*] [-a | --verbose] [-z | --xml-write] [-n | --no-xml-write] [-j | --objects *`objnames`*] [-J | --objects-file *`file`*] [-f | --file *`file`*] [-c | --check] [-S | --session-dir *`path`*] [-I | --include-dir *`path`*] [-O | --output-dir *`dir`*] [-x | --xml-style *`style`*] [-y | --dependency] [-i | --max-iteration *`limit`*] [-r | --max-recursion *`limit`*] [-g | --gzip] [-p | --deprecation *`number`*] [--failonwarn] [-M | --memory *`size`*] [--java-opts *`string`*] [-k | --noconf] [--logging *`string`*] [--logfile *`file`*] [-v | --version] [-h | --help] [-? | --usage] [*`template`*...]

Description

The **panc** command will compile a collection of pan language templates into a set of machine configuration files.

`-d, --debug`
: Enable the pan `debug` and `traceback` functions. Shorthand for `--debug-include '.*'`.

`--debug-include=`*`regex`*
: Define a pattern to selectively enable the pan `debug` and `traceback` functions. Those functions will be enabled for templates where the template name matches one of the include regular expressions *and* does not match an exclude regular expression. This option may appear multiple times.

`--debug-exclude=`*`regex`*
: Define a pattern to selectively disable the pan `debug` and `traceback` functions. Those functions will be disabled for templates where the template name matches one of the exclude regular expressions. This option may appear multiple times. Exclusion takes precedence over inclusion.

`--root-element=`*`nlist-dml`*
: A DML expression that evaluates to an nlist. This value will be used as the starting nlist for all object templates. This is a convenient mechanism for injecting build numbers and other metadata in the profiles.

`--dump-annotations`	Print the content of the annotations as they are processed. By default, they are not printed.
`--annotation-dir=`*`dir`*	Directory that will hold the files containing annotation information. The directory must exist. By default, the annotation information is not output.
`-a, --verbose`	At the end of a compilation, print run statistics including the numbers of files processed, total time, and memory used.
`-z, --xml-write`	Write machine configuration files (usually, but not exclusively, XML files). This is the default.
`-n, --no-xml-write`	Do not write machine configuration files.
`-j, --objects=`*`names`*	List what object templates to process and serialize to disk. This is a comma-separated list. The load path will be searched for these object templates.
`-J, --objects-file=`*`file`*	Read object templates to process from the given file. The file should contain one object template name per line.
`-f, --file=`*`file`*	File containing paths of templates to process. This file should contain one object template file name per line.
`-c, --check`	Only check the syntax of the given source templates; do not process into machine configuration files. This is a convenient and quick way to check that the syntax of all templates is correct before doing the more time-consuming complete processing.
`-S, --session-dir=`*`path`*	Set a session directory to search when looking for templates. The default is value is an empty list.
`-I, --include-dir=`*`path`*	Set which source directories to search when looking for templates. The value must be an absolute directory. This option can appear

	multiple times. If no values are given, the current working directory is used.
`-O, --output-dir=`*`dir`*	Set where the machine configuration files will be written. If this option is not specified, then the current working directory is used by default.
`-x, --xml-style=`*`style`*	Select the output style to use for the machine configuration files. This is usually an XML formatted file, but not always. Allowed values are "pan", "json", "text", and "dot". The default is value is "pan". The value "xmldb" is accepted, but deprecated.
`-y, --dependency`	Output the dependency information if this option is used. The dependency files will be named `*.xml.dep`, where the asterisk is replaced by the object template name.
`-i, --max-iteration=`*`limit`*	Set the limit on the maximum number of permitted loop iterations. This is used to avoid infinite loops. The default value is 5000.
`-r, --max-recursion=`*`limit`*	Set the limit on the maximum number of permitted recursions. The default value is 10.
`-g, --gzip`	If this option is used, then the generated machine configuration files will be compressed in gzip format.
`-p, --deprecation=`*`number`*	Set the value to use for deprecation warnings. The default is 0 indicating that warnings are emitted for features that will disappear in the next major release. Set to -1 to turn off all warnings. Use larger integers to see features that are planned to be deprecated in future releases.
`--failonwarn`	Treat warnings as errors.
`-M, --memory=`*`size`*	*(DEPRECATED)* Define size of memory to use for the java virtual machine. This is deprecated; use instead the `--java-opts`

	option with a value like "-Xmx1024M" for 1 GB of memory.
`--java-opts=`*`string`*	List of options to use when starting the java virtual machine. These are passed directly to the **java** command and must be valid. Multiple options can be specified by separating them with a space. When using multiple options, the full value must be enclosed in quotes.
`-k, --noconf`	Do not read any of the default configuration files. By default, the command searches for default options and arguments in the files `/etc/panc.conf` and `~/.panc.conf`. Options specified later take precedence. Options specified on the command line take precedence over values in the configuration files.
`--logging=`*`string`*	Enable compiler logging; possible values are "all", "none", "include", "call", "task", and "memory". A log file must be specified with the `--logfile` option to capture the logging information.
`--logfile=`*`file`*	Set the name of the file to use to store logging information.
`-v, --version`	Print the number of the compiler and exit.
`-h, --help`	Print a short summary of command usage.
`-?, --usage`	Print a short summary of command usage.

The **panc** command is just a wrapper script around the **java** command to simplify setting various options. The typical case is that the command is invoked without options and just a list of object templates as the arguments. Larger sets of templates will need to set the memory option for the Java Virtual Machine; this should be done through the `--java-opts` option.

Name

panc-build-stats.pl — create a report of panc build statistics

Synopsis

`panc-build-stats.pl` [`--help`] {logfile}

Description

The **panc-build-stats.pl** script will analyze a panc log file and report build statistics. The script takes the name of the log file as its only argument. If no argument is given or the `--help` option is used, a short usage message is printed. *The log file must have been created with "task" logging enabled.*

The script will extract the time required to execute, to set default values, to validate the configuration, to write the XML file, and to write a dependency file. It will also report the "build" time which is the time for executing, setting defaults, and validating an object file.

The analysis is written to the standard output, but may be saved in a file using standard IO stream redirection. The format of the file is appropriate for the R statistical analysis package, but should be trivial to import into excel or any other analysis package.

Example

If the output from the command is written to the file `build.txt`, then the following R script will do a simple analysis of the results. This will provide statistical results on the various build phases and show histograms of the distributions.

```
# R-script for simple analysis of build report
bstats <- read.table("build.txt")
attach(bstats)
summary(bstats)
hist(build, nclass=20)
hist(execute, nclass=20)
hist(execute, nclass=20)
hist(defaults, nclass=20)
hist(validation, nclass=20)
hist(xml, nclass=20)
hist(dep, nclass=20)
detach(bstats)
```

Name

panc-call-tree.pl — create a graph of pan call tree

Synopsis

`panc-call-tree.pl` [`--help`] [`--format=dot|hg`] {logfile}

Description

The **panc-call-tree.pl** script will analyze a panc log file and create a graph of the pan call tree. One output file will be created for each object template. The script takes the name of the log file as its only argument. If no argument is given or the `--help` option is used, a short usage message is printed. *The log file must have been created with "call" logging enabled.*

The graphs are written in either "dot" or "hypergraph" format. Graphviz [http://www.graphviz.org/] can be used to visualize graphs written in dot format. Hypergraph [http://hypergraph.sourceforge.net/] can be used to visualize graphs written in hypergraph format. Note that all "includes" are shown in the graph; in particular unique and declaration templates will appear in the graph wherever they are referenced.

Name

panc-compile-stats.pl — create a report of panc compilation statistics

Synopsis

`panc-compile-stats.pl` [`--help`] {logfile}

Description

The **panc-compile-stats.pl** script will analyze a panc log file and report compilation statistics. The script takes the name of the log file as its only argument. If no argument is given or the `--help` option is used, a short usage message is printed. *The log file must have been created with "task" logging enabled.*

The script will extract the start time of each compilation and its duration. This compilation is the time to parse a template file and create the internal representation of the template. The analysis is written to the standard output, but may be saved in a file using standard IO stream redirection. The format of the file is appropriate for the R statistical analysis package, but should be trivial to import into excel or any other analysis package.

Example

If the output from the command is written to the file `compile.txt`, then the following R script will create a "high-density" plot of the information. This graph shows a vertical line for each compilation, where the horizontal location is related to the start time and the height of the line the duration.

```
# R-script for simple analysis of compile report
cstats <- read.table("compile.txt")
attach(cstats)
plot(start/1000, duration, type="h", xlab="time (s)", ylab="duration (ms)")
detach(cstats)
```

Name

panc-memory.pl — create a report of panc memory utilization

Synopsis

`panc-memory.pl` [`--help`] {logfile}

Description

The **panc-memory.pl** script will analyze a panc log file and report on the memory usage. The script takes the name of the log file as its only argument. If no argument is given or the `--help` option is used, a short usage message is printed. *The log file must have been created with "memory" logging enabled.*

The script will extract the heap memory usage of the compiler as a function of time. The memory use is reported in megabytes and the times are in milliseconds. Usually one will want to use this information in conjunction with the thread information to understand the memory use as it relates to general compiler activity. Note that java uses sophisticated memory management and garbage collection techniques; fluctuations in memory usage may not be directly related to the compiler activity at any instant in time.

Example

If the output from the command is written to the file `memory.txt`, then the following R script will create a plot of the memory utilization as a function of time.

```
# R-script for simple analysis of memory report
mstats <- read.table("memory.txt")
attach(mstats)
plot(time/1000, memory, xlab="time (s)", ylab="memory (MB)", type="l")
detach(mstats)
```

Name

panc-profiling.pl — generate profiling information from panc log file

Synopsis

`panc-profiling.pl` [`--help`] [`--usefunctions`] {logfile}

Description

The **panc-profiling.pl** script will analyze a panc log file and report profiling information. The script takes the name of the log file as its first argument. The second argument determines if function call information will be included (flag=1) or not (flag=0). By default, the function call information is not included. If no argument is given or the `--help` option is used, a short usage message is printed. *The log file must have been created with "call" logging enabled.*

Two files are created for each object template: one with 'top-down' profile information and the other with 'bottom-up' information.

The top-down file contains a text representation of the call tree with each entry giving the total time spent in that template and any templates called from that template. At each level, one can use this to understand the relative time spent in a node and each direct descendant.

The bottom-up file provides how much time is spent directly in each template (or function), ignoring any time spent in templates called from it. This allows one to see how much time is spent in each template regardless of how the template (or function) was called.

All of the timing information is the "wall-clock" time, so other activity on the machine and the logging itself can influence the output. Nonetheless, the profiling information should be adequate to understand inefficient parts of a particular build.

Name

panc-threads.pl — create a report of thread activity

Synopsis

`panc-threads.pl` [`--help`] {logfile}

Description

The **panc-threads.pl** script will analyze a panc log file and report on build activity per thread. The script takes the name of the log file as its only argument. If no argument is given or the `--help` option is used, a short usage message is printed. *The log file must have been created with "task" logging enabled.*

The script will give the start time of build activity on any particular thread and the ending time. This can be used to understand the build and thread activity in a particular compilation. The times are given in milliseconds relative to the first entry in the log file.

Example

If the output from the command is written to the file `thread.txt`, then the following R script will create a plot showing the duration of the activity on each thread.

```
# R-script for simple analysis of thread report
tstats <- read.table("threads.txt")
attach(tstats)
plot(stop/1000,thread, type="n", xlab="time (s)", ylab="thread ID")
segments(start/1000, thread, stop/1000, thread)
detach(tstats)
```

APPENDIX D

Built-In Function Reference

Pan provides a large (and growing) number of built-in functions. These are treated as operators by the pan compiler implementation and are thus highly optimized. Consequently, they should be preferred to writing your own user-defined functions when possible. Because they are built into the compiler, the argument processing is different than that for user-defined functions. In particular, some arguments may be evaluated only when necessary and `null` can be a valid function argument.

Name

panc:append — adds a value to the end of a list

Synopsis

```
list append(value);

element value;

list append(target, value);

list target;
element value;

list append(target, value);

variable_reference target;
element value;
```

Description

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

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

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

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

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

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

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

Name

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

Synopsis

`string` **`base64_decode`**`(`*`encoded`*`);`

`string` *`encoded`*`;`

Description

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

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

Name

panc:base64_encode — encodes a string in base64 format

Synopsis

`string` **`base64_encode`**`(`*`unencoded`*`);`

`string` *`unencoded`*`;`

Description

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

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

Name

panc:clone — returns a clone (copy) of the argument

Synopsis

```
element clone(arg);

element arg;
```

Description

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

Name

panc:create — create an nlist from a structure template

Synopsis

```
nlist create(tpl_name, );

string tpl_name;
...;
```

Description

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

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

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

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

Name

panc:debug — print debugging information to the console

Synopsis

```
string debug(msg);

string msg;
```

Description

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

Name

panc:delete — delete the element identified by the variable expression

Synopsis

```
undef delete(arg);

variable_expression arg;
```

Description

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

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

Name

panc:deprecated — print deprecation warning to console

Synopsis

```
string deprecated( level, msg);

long level;
string msg;
```

Description

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

Name

panc:digest — creates a digest of a message using the specified algorithm

Synopsis

```
string digest(algorithm, message);

string algorithm;
string message;
```

Description

This function returns a digest of the message using the specified algorithm. The valid algorithms are: `MD2`, `MD5`, `SHA`, `SHA-1`, `SHA-256`, `SHA-384`, and `SHA-512`. The algorithm name is not case sensitive.

Name

panc:error — print message to console and abort compilation

Synopsis

```
void error(msg);

string msg;
```

Description

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

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

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

  # normal processing...
};
```

Name

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

Synopsis

```
string escape(str);

string str;
```

Description

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

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

Name

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

Synopsis

```
boolean exists(var);

variable_expression var;

boolean exists(path);

string path;

boolean exists(tpl);

string tpl;
```

Description

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

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

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

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

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

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

Name

panc:file_contents — provide contents of file as a string

Synopsis

```
string file_contents(filename);

string filename;
```

Description

This function will return a string containing the contents of the named file. The file is located using the standard source file lookup algorithm. Because the load path is used to find the file, this function may not be used to create a compile-time constant. If the file cannot be found, an error will be raised.

Name

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

Synopsis

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

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

Description

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

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

An example of using `first` with an nlist:

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

Name

panc:format — format a string by replacing references to parameters

Synopsis

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

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

Description

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

Name

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

Synopsis

```
string|undef if_exists(tpl);

string tpl;
```

Description

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

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

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

Name

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

Synopsis

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

string sub;
string arg;
long start;

long index(sub, list, start);

property sub;
string list;
long start;

string index(sub, arg, start);

property sub;
nlist arg;
long start;

long index(sub, arg, start);

nlist sub;
list arg;
long start;

string index(sub, arg, start);

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

Description

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

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

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

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

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

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

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

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

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

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

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

```
# search a record in a list of records (result = 1, the second nlist)
'/ll1' = index(
              nlist('key', 'foo'),
              list(
                   nlist('key', 'bar', 'val', 101),
                   nlist('key', 'foo')
                  )
             );

# search a record in a list of records starting at index (result = 1, the second nlist)
'/ll2' = index(
              nlist('key', 'foo'),
              list(
                   nlist('key', 'bar', 'val', 101),
                   nlist('key', 'foo'),
                   nlist('key', 'bar', 'val', 101),
                   nlist('key', 'foo'),
                   nlist('key', 'bar', 'val', 101),
                   nlist('key', 'foo')
                  ),
              1
             );
```

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

```
# search for matching nlist (result = 'b')
```

```
'/nn1' = index(
              nlist('key', 'foo'),
              nlist(
                   'a', nlist('key', 'bar', 'val', 101),
                   'b', nlist('key', 'foo')
                  )
             );

# skip first match and return index of second match (result='d')
'/nn2' = index(
              nlist('key', 'foo'),
              nlist(
                   'a', nlist('key', 'bar', 'val', 101),
                   'b', nlist('key', 'foo'),
                   'c', nlist('key', 'bar', 'val', 101),
                   'd', nlist('key', 'foo'),
                   'e', nlist('key', 'bar', 'val', 101),
                   'f', nlist('key', 'foo')
                  ),
              1
             );
```

Name

panc:is_boolean — checks to see if the argument is a double

Synopsis

```
boolean is_boolean(arg);

element arg;
```

Description

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

Name

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

Synopsis

```
boolean is_defined(arg);

element arg;
```

Description

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

Name

panc:is_double — checks to see if the argument is a double

Synopsis

```
boolean is_double(arg);

element arg;
```

Description

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

Name

panc:is_list — checks to see if the argument is a double

Synopsis

```
boolean is_list(arg);

element arg;
```

Description

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

Name

panc:is_long — checks to see if the argument is a long

Synopsis

```
boolean is_long(arg);

element arg;
```

Description

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

Name

panc:is_nlist — checks to see if the argument is an nlist

Synopsis

```
boolean is_nlist(arg);

element arg;
```

Description

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

Name

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

Synopsis

```
boolean is_null(arg);

element arg;
```

Description

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

Name

panc:is_number — checks to see if the argument is a number

Synopsis

```
boolean is_number(arg);

element arg;
```

Description

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

Name

panc:is_property — checks to see if the argument is a property

Synopsis

```
boolean is_property(arg);

element arg;
```

Description

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

Name

panc:is_resource — checks to see if the argument is a resource

Synopsis

```
boolean is_resource(arg);

element arg;
```

Description

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

Name

panc:is_string — checks to see if the argument is a string

Synopsis

```
boolean is_string(arg);

element arg;
```

Description

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

Name

panc:key — returns name of child based on the index

Synopsis

```
string key(resource, index);

nlist resource;
long index;
```

Description

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

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

'/keys' = {

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

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

Name

panc:length — returns size of a string or resource

Synopsis

```
long length(str);

string str;

long length(res);

resource res;
```

Description

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

Name

panc:list — create a new list consisting of the function arguments

Synopsis

```
list list(elem, );

element elem;
...;
```

Description

Returns a newly created list containing the function arguments.

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

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

Name

panc:match — checks if a regular expression matches a string

Synopsis

```
boolean match(target, regex);

string target;
string regex;
```

Description

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

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

Name

panc:matches — checks if a regular expression matches a string

Synopsis

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

string target;
string regex;
```

Description

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

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

Name

panc:merge — combine two resources into a single one

Synopsis

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

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

Description

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

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

Name

panc:nlist — create an nlist from the arguments

Synopsis

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

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

Description

The `nlist` function returns a new nlist consisting of the passed arguments; the arguments must be key value pairs. All of the keys must be strings and have values that are legal path terms (see Path Literals Section).

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

Name

panc:next — increment iterator over a resource

Synopsis

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

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

Description

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

Name

panc:path_exists — determines if a path exists

Synopsis

```
boolean path_exists(path);

string path;
```

Description

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

Name

panc:prepend — adds a value to the beginning of a list

Synopsis

```
list prepend(value);

element value;

list prepend(target, value);

list target;
element value;

list prepend(target, value);

variable_reference target;
element value;
```

Description

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

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

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

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

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

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

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

Name

panc:replace — replace all occurrences of a regular expression

Synopsis

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

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

Description

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

Name

panc:return — exit DML block with given value

Synopsis

```
element return(value);

element value;
```

Description

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

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

Name

panc:splice — insert string or list into another

Synopsis

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

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

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

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

Description

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

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

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

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

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

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

Important

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

Name

panc:split — split a string using a regular expression

Synopsis

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

string regex;
string target;

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

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

Description

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

Name

panc:substr — extract a substring from a string

Synopsis

```
string substr(target, start);

string target;
long start;

string substr(target, start, length);

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

Description

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

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

Name

panc:to_boolean — convert argument to a boolean value

Synopsis

```
boolean to_boolean(prop);

property prop;
```

Description

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

Name

panc:to_double — convert argument to a double value

Synopsis

```
double to_double(prop);

property prop;
```

Description

This function converts the given property into a double.

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

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

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

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

Name

panc:to_long — convert argument to a long value

Synopsis

```
long to_long(prop);

property prop;
```

Description

This function converts the given property into a long value.

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

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

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

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

Name

panc:to_lowercase — change all uppercase letters to lowercase

Synopsis

```
string to_lowercase(target);

string target;
```

Description

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

Name

panc:to_string — convert argument to a string value

Synopsis

```
string to_string(elem);

element elem;
```

Description

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

Name

panc:to_uppercase — change all lowercase letters to uppercase

Synopsis

```
string to_uppercase(target);

string target;
```

Description

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

Name

panc:traceback — print message and traceback to console

Synopsis

```
string traceback(msg);

string msg;
```

Description

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

Name

panc:unescape — replaces escaped characters with ASCII characters

Synopsis

```
string unescape(str);

string str;
```

Description

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

Name

panc:value — retrieve a value specified by a path

Synopsis

```
element value(path);

string path;
```

Description

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

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