# Pan Language Compiler Documentation

*Release 10.2-SNAPSHOT*

**Quattor Collaboration**

**Nov 27, 2018**

# Contents

1 Release Notes 1

2 Reference Manual 7

3 Appendices 55

4 Search 97

CHAPTER 1

# Release Notes

## 1.1 Release Notes

This document the release notes for the pan language compiler release notes as well as a detailed change log. See the full documentation for information about the pan language and use of the compiler.

### 1.1.1 License

Licensed under the Apache License, Version 2.0 (the "License"). You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

### 1.1.2 Support

The v10 series is in active development; the v9 series is frozen. Both of these series are supported.

**The old v8 series of releases is no longer supported.** Migration from v8 to a more recent release is very strongly recommended.

### 1.1.3 Upcoming Changes

The following upcoming and potentially breaking changes are scheduled for the next major pan compiler release (v11):

- Support for the file extension '.tpl' is deprecated and will be removed in the v11 series of the compiler. Rename your pan source files from '*.tpl' to* '.pan'.
- The path '/panc' will be reserved by the compiler. Values under this path will be used to transmit information about the pan compiler version and compilation flags to downstream tools. Avoid putting configuration information in the '/panc' part of the configuration tree.

### 1.1.4 Change Log

#### Version 10.4

- (GitHub Issue #118) Built-in join function.
- (GitHub Issue #121) Implement choice type.
- (GitHub Issue #122) Formatting.
- (GitHub Issue #123) Implement built-in validate function to check type.
- (GitHub Issue #124) Working initial-data command line option.
- (GitHub Issue #125) Standalone pan tests.
- (GitHub Issue #126) Support booleans with (in)equality operator.

#### Version 10.3

- (GitHub Issue #105) Resource protection should be deep, not shallow.
- (GitHub Issue #107) Fix error message for undefined values.

#### Version 10.2

- (GitHub Issue #80) enable automated build of documentation on the http://quattor-pan.readthedocs.org/.
- (GitHub Issue #79, #81) move documentation sources from DocBook to restructured text.
- (GitHub Issue #76) remove 'object' template reference in output stats
- (GitHub Issue #71, #72, #73, #74, #75) add options for displaying the pan compiler version and update documentation
- (GitHub Issue #70) create a null formatter mainly for performance testing

#### Version 10.1

- (GitHub Issue #68) revert a couple of UTF-8 read/write changes to conserve backward compatibility
- (GitHub Issue #67) update source and bytecode to java 1.6
- (GitHub Issue #13) change nlist references to dict
- (GitHub Issue #48) allow variable substitution for bind/valid paths
- (GitHub Issue #34) add the file_exists function
- (GitHub Issue #63) allow user to specify number of threads for processing (nthread option)
- (GitHub Issue #61) fix processing of include path CLI argument
- (GitHub Issue #59) add substitute function to replace named values in string template
- (GitHub Issue #54) convert source files to UTF-8
- (GitHub Issue #49) add warning in docs that all pan source files must be UTF-8 (also for file_contents function)
- (GitHub Issue #47) fix compiler hang when using escape sequence in path literal
- (GitHub Issue #43) fix compiler crash when SELF is used as a function

- (GitHub Issue #41) RPM package should not own /usr/bin and /usr/lib
- (GitHub Issue #40) ensure line number and file name are correct for traceback function
- (GitHub Issue #38) add ip address and netmask functions
- (GitHub Issue #37) ensure line numbers appear in error message for bad default values
- (GitHub Issue #36) allow to_long to treat values like "08" and "09"
- (GitHub Issue #31) fix options processing for CLI (bad processing causes failure)
- (GitHub Issue #29) update links in documentation to GitHub from SourceForge
- (GitHub Issue #15, #24) add OBJECT to debug and error output
- (GitHub Issue #31) panc command line fails

### Version 10.0

- (GitHub Issue #27) Remove session directory functionality
- (GitHub Issue #5) Remove deprecated options from panc ant task
- (GitHub Issue #4) Remove panc-old script
- (GitHub Issue #2) Remove deprecation level attribute in favor of warnings attribute in pan-syntax-check mojo
- (GitHub Issue #26) Restore backward compatibility for gzip output flag

### Version 9.3

No additional changes besides those in RC1 and RC2.

### Version 9.3-RC2

- (SF Bug #3585672) Permit both lower and upper case strings for warnings flag in ant and maven tasks.
- (SF Bug #3585346) Misleading deprecation message for debug element

### Version 9.3-RC1

- (SF Bug #3582159) Uncaught exception when creating XML transformation
- (SF RFE #3581805) Remove support for XMLDB format.
- (SF RFE #3581801) Change dependency file extension from *.xml.dep to *.dep.
- (SF RFE #3535682) Allow multiple output formats to be generated from CLI.
- (SF Bug #3535413) Check timestamps of all requested output file formats.
- (SF Bug #3529737) Non-object templates can be accessed via value().
- (SF Bug #3579769) Tests failed because of change in TreeSet contract in Java 1.7.
- (SF Bug #3579770) Shell scripts use bash syntax. Explicitly use bash in she-bang lines.
- (SF Bug #3581163) Invalid replacement string in replace() raises uncaught exception.
- (SF RFE #3489988) Allow negative values in range expressions.
- The include syntax without required braces is now allowed.

- The `panc` command no longer includes the possibility to process annotations. This functionality is now in a separate command `panc-annotations`.
- The `panc` command now uses a streamlined set of options that are not compatible with the previous one. The previous one can be invoked with the `panc-old` command.

### Version 9.2

- (SF RFE #3489506) Provide a pan maven archetype. A rudimentary implementation is available which uses the panc maven plugin.
- (SF RFE #3489504) Provide a maven build mojo. A rudimentary implementation is available in the panc maven plugin.
- (SF RFE #3489048) Switch unit tests to use the pan XML format instead of xmldb.
- (SF RFE #3489084) Remove support for panx extension. This has been removed as an XML input format is no longer in the roadmap.
- (SF RFE #3477756) Provide JSON output option. Initial JSON formatter is available; detailed serialization may change based on feedback. The pan compiler now includes the GSON library (Apache 2 license) to handle the JSON serialization.
- (SF RFE #3477753) Deprecate xmldb format. Use the standard pan XML format instead of the xmldb format.
- (SF Bug #3488948) Annotation information in pan book is inaccurate. The description has been correct and expanded somewhat.

### Version 9.1

- (SF Bug #3485801) pan does not build on Windows; full build and unit tests now run correctly on windows
- (SF Bug #3485492) `file_contents` does not work correctly on Windows; problems with file name handling have been resolved
- (SF Bug #3483938) Fix the README file to contain information on changes up through the production 9.0 release.

### Version 9.0

Production release contains the same features as RC3. All version numbers will be considered production releases unless marked explicitly as alpha, beta, or release candidates.

### Version 9.0.0-RC3

- (SF RFE #3422390) The root element used as the starting point for all machine profiles can be specified from the command line and ant task. This allows the injection of data into all of the profiles without having to include explicitly a template in all machine profiles. This will be useful for injecting build metadata into the profiles. Note that the injected data must still follow the global schema (if defined), otherwise builds will fail with validation errors.

### Version 9.0.0-RC2

The documentation has been significantly reorganized with all of the documentation apart from this README combined into a single “pan book”.

### Version 9.0.0-RC1

This release contains the following changes:

- (SF Bug #3171788) Improve error message for format() function when there is a mismatch between given format string and arguments.
- (SF RFE #3386906) Support for \b (backspace) and \f (form feed) escape sequences in double-quoted strings.
- (SF Bug #3186921) Dependency calculation in ant task does not work correctly for namespaced object templates.

# CHAPTER 2

# Reference Manual

## 2.1 Pan Language

Comprehensive overview of the pan language and the pan language compiler.

### 2.1.1 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 :ref:‘running_panc‘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 releases area of the GitHub repository.

The pan compiler requires a Java Runtime Environment (JRE) or Java Development Kit (JDK) 1.6 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.

> **Warning:** The GNU Java Compiler (GJC) distributed with many Linux-based operating systems is **not** a certified version of the Java Virtual Machine. 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=$PANC_HOME/bin:$PATH
```

The above will work for Bourne shells on *nix-like operating systems; adjust the command for the operating system and 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 command

```
$ panc --help
```

This gives a complete list of all of the available options. If the 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
```

Should give the following:

```
<?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 that was written to the `hello_world.json` file.

```
$ panc --formats json hello_world.pan
$ cat hello_world.json
```

Should give the following:

```
{
  "message": "Hello World!"
}
```

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

```
$ panc --formats text hello_world.pan
$ cat hello_world.txt
```

Should give the following:

```
+-profile
  $ message : (string) 'hello'
```

The output file is named `hello_world.txt`. It provides the same information as the other formats, but is easier to read.

The last style is the "dot" format.

```
$ panc --formats dot hello_world.pan
$ cat hello_world.dot
```

Should give the following:

```
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 to generate a graph of the configuration. Processing the above file with Graphviz produces the image shown in the *Graph of hello_world.pan configuration*.

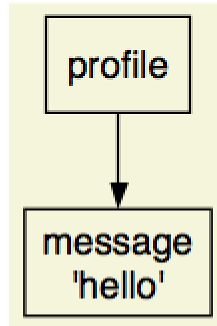


Fig. 1: Graph of `hello_world.pan` configuration

### 2.1.2 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 --formats 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
```

(continues on next page)

(continued from previous page)

```
      $ 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 --formats 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;

type batch_capabilities = string[];

type batch_queue_list = string[1..];

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

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

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

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

The `batch_worker` type defines a record (dict 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.

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

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.

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.

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.

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

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')
```

(continues on next page)

(continued from previous page)

```
  'capabilities' ? batch_capabilities
};

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

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

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

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'.

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

By default, a queue will be enabled.

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

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

function valid_batch_node_dict = {
  foreach (hostname; properties; ARGV[0]) {
    path = 'profiles/' + hostname + ':/batch/worker';
    if (!path_exists(path)) {
      error(path + ' doesn''t exist');
      return(false);
    };
  };
  true;
};

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

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

(continues on next page)

(continued from previous page)

```
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)
  'capabilities' ? batch_capabilities
};

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

type batch_server = {
  'nodes' : batch_node{} with valid_batch_node_dict(SELF)
  'queues' : batch_queue{} = dict('default', dict())
};

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

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.

The argument to this function is the dict 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.

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.

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.

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:

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

Run the `valid_batch_node_dict` function for the server's node dict.

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.

The `prefix` statement itself also supports a relative path. In that case, a previously defined `prefix` statement with an absolute path is required, and the relative prefix is relative to the (last) absolute prefix. This is very useful when dealing with very large paths and a complex schema.

The previous example could be rewritten using relative prefixes as follows

```
...

prefix '/batch/server/nodes';

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

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

## 2.1.3 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.

> **Warning:** All pan source files, templates as well as included files, must be encoded in UTF-8. No other character encodings are supported.

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 `.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/`.

> **Warning:** The older file extension "tpl" is also accepted by the pan compiler, but is **deprecated**. Support for this older prefix will disappear in the next major release. Currently, 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 dicts 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.

The path used in these statements can contain *global variable* references of the form `${vname}`. This allows the binding between the validation code and a path to be determined when a profile is built. If the path references an undefined global variable, then will abort with an error. The build will also be aborted if the path is not valid after the values of the variables have been substituted. Below is an example of using *global variable* references:

```
variable MYFILE = 'test';
bind  '/a/${MYFILE}' = type-spec;
```

As with any path element, the variable contents can be escaped if necessary by enclosing the variable reference into '{}'. For example:

```
variable MYFILE = '/tmp/test';
bind  '/a/{${MYFILE}}' = type-spec;
```

See chapter on Validation for more details.

### 2.1.4 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 dict. The 'list' is an ordered list of elements, which uses the index (an integer) as the key. The named list (dict) 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 the graph *Pan language type hierarchy*, including the two special types `undef` and `null`.

#### 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).

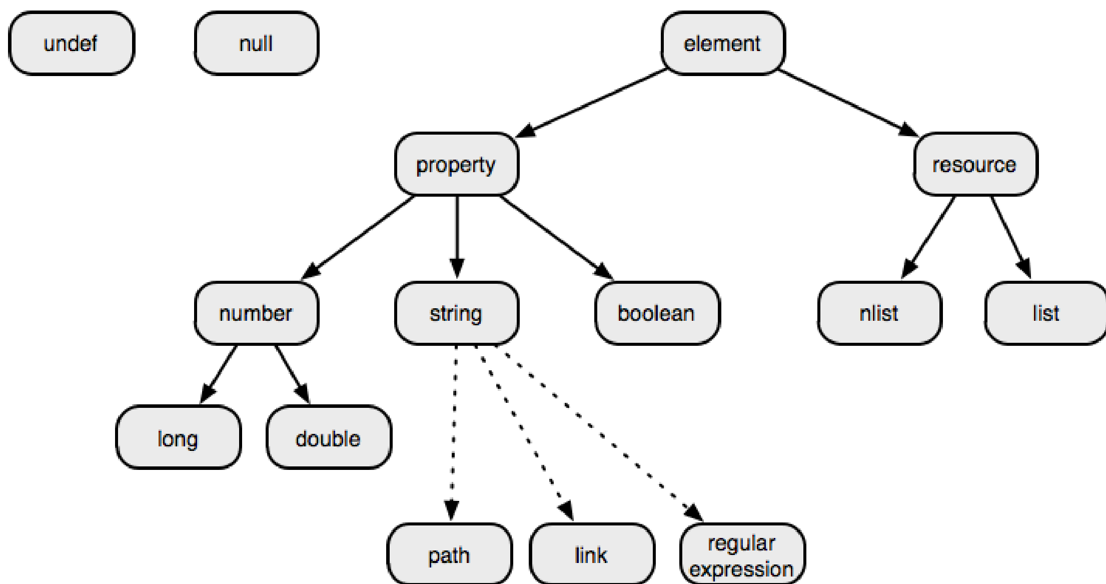


Fig. 2: Pan language type hierarchy

### 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 curly braces. For example, the following creates an absolute path with three terms:

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

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

Another example with a DML block:

```
"/alpha" = dict("{a/b}", "delta");
```

This is is equivalent to:

```
"/alpha" = dict(escape("a/b"), "delta");
```

### 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 dict. 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 dict (named list) associates a name with an element; these are also known as hashes or associative arrays. One dict in the above example is `/hardware/cpu`, which has `arch`, `cores`, `model`, `number`, and `speed` as children. Note that the order of an dict 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 dict 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 dicts 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 one child:

```
object template mylist;

'/mylist/0' = 'decimal index';
```

The indices can be specified in decimal (digits only). Leading `` 0``(s) (e.g. octal notation) are considered ambiguous and as such invalid. The names of children in an dict ``

can be any string that has at least one non-digit character, starts with a so-called ``word-character (a-zA-Z0-9_) and is followed by (zero or more) word-characters or +-..

### 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 to which it is assigned. Most operations and functions will report an error if this value is processed directly.

## 2.1.5 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 dict
'/table/red' = 'rouge';
'/table/green' = 'vert';

'/test' = {
  x = list('a', 'b', 'c'); # x is a list
  y = value('/table');     # y is a dict
  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.

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

Table: Unary 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 |
| == | boolean | equal |
| == | string | lexical equal |
| != | number | not equal |
| != | boolean | 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: Binary DML Operators

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

Table: Operator Precedence (lowest to highest)

### 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 `dict`).

> **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 dict. 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 dict). 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 dicts.

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 dict), `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.

### 2.1.6 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.

| Name | Description |
|---|---|
| *file_contents* | Lookup the named file and provide the file's contents as a string. |
| *file_exists* | Lookup the named file and return true if it exists; return false otherwise. |
| *format* | Generate a formatted string based on the formatting parameters and the values provided. |
| *index* | Return the index of a substring or -1 if the substring is not found. |
| *length* | Gives the length of a string. |
| *match* | Return a boolean indicating if a string matches the given regular expression. |
| *matches* | Return an array containing the matched string and matched groups for a given string and regular expression. |
| *replace* | Replace all occurrences of a substring within a given string. |
| *splice* | Remove a substring and optionally replace it with another. |
| *split* | Split a string based on a given regular expression and return an array of the results. |
| *substitute* | Substitute named values in a string template. |
| *substr* | Extract a substring from the given string. |
| *to_lowercase* | Change all of the characters in a string to lowercase (using the US locale). |
| *to_uppercase* | Change all of the characters in a string to uppercase (using the US locale). |

Table: String Manipulation Functions

| Name | Description |
|---|---|
| *debug* | Print a debugging message to the standard error stream. Returns the message or `undef`. |
| *error* | Print an error message to the standard error and terminate processing. |
| *traceback* | Print an error message to the standard error along with a traceback. Returns `undef`. |
| *deprecated* | Print a warning message to the standard error if required by the deprecation level in effect. Returns ‘‘the message or undef‘‘. |

Table: Debugging Functions

| Name | Description |
|---|---|
| *base64_decode* | Decode a string that is encoded using the Base64 standard. |
| *base64_encode* | Encode a string using the Base64 standard. |
| *digest* | Create message digest using specified algorithm. |
| *escape* | Escape characters within the string to ensure string is a valid dict key (path term). |
| *unescape* | Transform an escaped string into its original form. |
| *json_decode* | Convert a JSON-encoded string to a PAN data structure. |
| *json_encode* | Convert a PAN data structure to a JSON-encoded string. |

Table: Encoding and Decoding Functions

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

Table: Resource Manipulation Functions

| Name | Description |
|---|---|
| *is_boolean* | 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* | 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* | 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* | 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* | 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_dict* | Check if the argument is an dict. 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* | 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* | 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* | 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* | Check if the argument is a list or dict. 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* | 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: Type Checking Functions

| Name | Description |
|---|---|
| *to_boolean* | 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* | 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* | 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. An optional second argument can be provided that defines the radix to use. |
| *to_string* | Convert the argument to a string. The function will return a string representation for any argument, including list and dict. |
| *ip4_to_long* | Convert the argument, which must be a string representing an IPv4 address in dotted notation to a long. |
| *long_to_ip4* | Convert the argument, a long into an IPv4 address in numbers-and-dots notation |

Table: Type Conversion Functions

| Name | Description |
|---|---|
| *clone* | Create a deep copy of the given value. |
| *delete* | Delete a local variable or child of a local variable. |
| *exists* | Return true if the given argument exists. The argument can either be a variable reference, path, or template name. |
| *join* | Returns a string with the arguments joined, using the the passed delimeter. |
| *path_exists* | Return true if the given path exists. The argument must be an absolute or external path. |
| *if_exists* | 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. |
| *is_valid* | This function checks whether a certain element meets the requirements of a certain type. |
| *return* | 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* | Retrieve the value associated with the given path. The path may either be an absolute or external path. |

Table: Miscellaneous Functions

### 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  = 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';

};
```

### 2.1.7 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_dict = double{};
type small_even_dict = 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 dict 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 dict 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 dicts 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 dict 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.

### The choice-type

The choice type is an advanced type that can be used to create user-defined types. It takes one or more strings as input arguments. These arguments represent the possible choices that can be assigned to a certain path. For example:

```
type mychoice = choice("a", "b", "c");
```

defines a type with only three possible choices. The choice type can be used in the same way and in combination with the above explained types.

### 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');
      };

    };

  };
```

(continues on next page)

(continued from previous page)

```
};

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_dict = {

  # 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_dict = component{} with valid_component_dict(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.

## 2.1.8 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
```

(continues on next page)

(continued from previous page)

```
  '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;
```

(continues on next page)

(continued from previous page)

```
'/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 dict 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;
```

(continues on next page)

(continued from previous page)

```
'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.

### 2.1.9 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 mysite/example;

@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 `--output-dir` option, as well as the root file system directory for all of the processed files with the `--base-dir` option if this is not the current directory. `--base-dir` option must be adjusted so that the template file paths specified match the template namespaces, as for compiling the templates.

Below is an example

```
$ panc-annotations \
          --output-dir=annotations \
          --base-dir=templates \
          mysite/example.pan
```

This command will produce the following output (with whitespace and indentation added for clarity) in the file `example.pan.annotation.xml` located in `annotations/mysite` from template file `example.pan` located in subdirectory `templates/mysite` of current directory.

```
<?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"/>
```

(continues on next page)

(continued from previous page)

```
            <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 dict. For example, this expression

```
dict('build-metadata', dict('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.

### 2.1.10 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' = dict('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' = dict(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.

## 2.1.11 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;
```

(continues on next page)

(continued from previous page)

```
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;

```

(continues on next page)

(continued from previous page)

```
'/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.

### 2.1.12 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

**Q:** "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.

**Q:** 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.

**Q:** "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.

**Q:** 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 in the issues area of GitHub.

CHAPTER 3

# Appendices

## 3.1 Standard Functions

Pan provides a large (and growing) number of standard 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.

### 3.1.1 append

#### Name

append – adds a value to the end of a list

#### Synopsis

list **append** (element *value*)

list **append** (list *target*, element *value*)

list **append** (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);
};
```

### 3.1.2 base64_decode

#### Name

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

#### Synopsis

string **base64_decode** (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=');
```

### 3.1.3 base64_encode

#### Name

base64_encode – encodes a string in base64 format

#### Synopsis

string **base64_encode** (string *encoded)

#### 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');
```

### 3.1.4 clone

#### Name

clone – returns a clone (copy) of the argument

#### Synopsis

element **clone** (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.

### 3.1.5 create

#### Name

create – create a dict from a structure template

#### Synopsis

dict **create** (string *tpl_name*, . . . )

#### Description

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

```
# 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';
```

### 3.1.6 debug

#### Name

debug – print debugging information to the console

#### Synopsis

string **debug** (string *msg*)

string **debug** (string *fmt*, element *param*, . . . )

#### Description

This function will print the given string to the console (on stdout) and return the message as the result. The function also accepts format strings, similar to the `format` function. The string has '[object] ' prepended to it, where 'object' is the name of the object template. 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.

### 3.1.7 delete

#### Name

delete – delete the element identified by the variable expression

#### Synopsis

undef **delete** (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;
};
```

### 3.1.8 deprecated

#### Name

deprecated – print deprecation warning to console

#### Synopsis

string **deprecated** (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.

### 3.1.9 dict

#### Name

dict – create an dict from the arguments

#### Synopsis

dict **dict** (string *key*, element *property*, . . . )

#### Description

The `dict` function returns a new dict 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 dict associates name with long value
'/result' = dict(
  'one', 1,
  'two', 2,
```

(continues on next page)

(continued from previous page)

```
  'three', 3,
};
```

### 3.1.10 digest

#### Name

digest – creates a digest of a message using the specified algorithm

#### Synopsis

string **digest** (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.

### 3.1.11 error

#### Name

error – print message to console and abort compilation

#### Synopsis

void **error** (string *msg*)

void **error** (string *fmt*, element *param*, . . . )

#### Description

This function prints the given message to the console (stderr) and aborts the compilation. The function also accepts format strings, similar to the `format` function. The message has '[object ]' prepended to it as a convenience. 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...
};
```

### 3.1.12 escape

#### Name

escape – escape non-alphanumeric characters to allow use as dict key

#### Synopsis

string **escape** (string *str*)

#### Description

This function escapes non-alphanumeric characters in the argument so that it can be used inside paths, for instance as an dict 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');
```

### 3.1.13 exists

#### Name

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

#### Synopsis

boolean **exists** (variable_expression *var*)

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

boolean **exists** (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.

## 3.1.14 file_contents

### Name

file_contents – provide contents of file as a string

### Synopsis

string **file_contents** (string *filename*)

string **file_contents** (string *filename*, string *compression*)

### 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.

Optional second argument indication what type of compression is used in the file, and the contents will be decompressed. Supported compression is *gzip*.

## 3.1.15 file_exists

### Name

file_exists – determine if the named file exists

### Synopsis

string **file_exists** (string *filename*)

### Description

This function will return a boolean indicating whether the named file exists. 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.

## 3.1.16 first

### Name

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

### Synopsis

**boolean first (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 dict 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 dict:

```
# put the list of all the keys of table inside keys
table = dict("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")
```

### 3.1.17 format

#### Name

format – format a string by replacing references to parameters

#### Synopsis

string **format** (string *fmt*, element *param*, . . . )

#### Description

The `format` function will replace all references within the `fmt` string with the values of the referenced elements. 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. When passing a resource as an argument, the string replacement field should be used.

### 3.1.18 if_exists

#### Name

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

#### Synopsis

stringlundef **if_exists** (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.

### 3.1.19 index

#### Name

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

#### Synopsis

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

long **index** (property *sub*, string *list*, long *start*)

string **index** (property *sub*, dict *arg*, long *start*)

long **index** (dict *sub*, list *arg*, long *start*)

string **index** (dict *sub*, dict *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' = dict('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 dict inside the given list of dicts 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 dict)
'/ll1' = index(
              dict('key', 'foo'),
              list(
                   dict('key', 'bar', 'val', 101),
                   dict('key', 'foo')
                  )
             );

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

The last form searches for the given dict inside the given dict of dicts 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 dict (result = 'b')
'/nn1' = index(
              dict('key', 'foo'),
              dict(
                   'a', dict('key', 'bar', 'val', 101),
                   'b', dict('key', 'foo')
                  )
             );

# skip first match and return index of second match (result='d')
'/nn2' = index(
              dict('key', 'foo'),
```

(continues on next page)

(continued from previous page)

```
            dict(
                'a', dict('key', 'bar', 'val', 101),
                'b', dict('key', 'foo'),
                'c', dict('key', 'bar', 'val', 101),
                'd', dict('key', 'foo'),
                'e', dict('key', 'bar', 'val', 101),
                'f', dict('key', 'foo')
            ),
            1
        );
```

### 3.1.20 ip4_to_long

#### Name

ip4_to_long – converts an IP address in dotted format with an optional bitmask to a list of longs

#### Synopsis

long[] **ip4_to_long** (string *ip*)

#### Description

The `ip4_to_long` function returns the binary representation of an IPv4 address or network specification represented as a dotted string, where the netmask part is optional, like `inet_aton` does in the C standard library.

The first element of the return value is the binary representation of the IP address, where the second, if present, is the binary representation of the network mask.

This can be used for applying network masks and calculating network ranges.

```
variable NETWORK_RANGE_FOR_LOCALHOST = {
    l = ip4_to_long("127.0.0.1/8");
    l[0] & l[1];
};
```

```
variable BINARY_LOCALHOST = ip4_to_long("127.0.0.1");
```

### 3.1.21 is_boolean

#### Name

is_boolean – checks to see if the argument is a double

#### Synopsis

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

#### Description

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

### 3.1.22 is_defined

#### Name

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

#### Synopsis

boolean **is_defined** (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.

### 3.1.23 is_double

#### Name

is_double –checks to see if the argument is a double

#### Synopsis

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

#### Description

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

### 3.1.24 is_list

#### Name

is_list – checks to see if the argument is a double

#### Synopsis

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

#### Description

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

### 3.1.25 is_long

#### Name

is_long – checks to see if the argument is a long

#### Synopsis

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

#### Description

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

### 3.1.26 is_dict

#### Name

is_dict – checks to see if the argument is an dict

#### Synopsis

boolean **is_dict** (element *arg*)

#### Description

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

### 3.1.27 is_null

#### Name

is_null – checks to see if the argument is null

#### Synopsis

boolean **is_null** (element *arg*)

#### Description

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

### 3.1.28 is_number

#### Name

is_number – checks to see if the argument is a number

#### Synopsis

boolean **is_number** (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.

### 3.1.29 is_property

#### Name

is_property – checks to see if the argument is a property

#### Synopsis

boolean **is_property** (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.

### 3.1.30 is_resource

#### Name

is_resource – checks to see if the argument is a resource

#### Synopsis

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

#### Description

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

### 3.1.31 is_string

#### Name

is_string – checks to see if the argument is a string

#### Synopsis

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

#### Description

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

### 3.1.32 is_valid

#### Name

is_valid – checks if an element meets the requirements of a certain type

#### Synopsis

boolean **is_valid** (type *type*, element *el*)

#### Description

This function checks whether a certain element meets the requirements of a certain type. The argument can be a variable or an operation, since these eventually will lead to a certain value. The function can be used as follows:

```
type mytype = string(2..);
variable X = "Message";

'/result' = is_valid(mytype, X);
```

In this case `'/result'` will be of type boolean and hold `true` as a value.

### 3.1.33 join

#### Name

join – joins the passed arguments

#### Synopsis

string **join** (string *delimeter*, list *resource*)

string **join** (string *delimeter*, string *arg1*, string *arg2*, . . . )

#### Description

This function takes a delimeter and a list of strings, or each of the strings individually, and joins them with the given delimeter. Only (a list of) strings can be passed as arguments.

```
# joining a list
'/x' = list("a", "b", "c");
'/rx' = join("-", value('/x')); # This will return "a-b-c"

# joining individual arguments
'/rx' = join("-", "a", "b", "c"); # This will also return "a-b-c"
```

### 3.1.34 key

#### Name

key – returns name of child based on the index

#### Synopsis

string **key** (dict *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 dict. The index corresponds to the key's position in the list of all keys, sorted in lexical order. The first index is 0.

```
'/table' = dict('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 '
```

### 3.1.35 json_decode

#### Name

json_decode – convert a JSON-encoded string to a PAN data structure

#### Synopsis

element **json_decode** (string *encoded*)

#### Description

The `json_decode` function parses the JSON-encoded argument into the appropriate PAN data structure. The argument can be either a primitive JSON type, a list or a JSON object. In case of JSON objects, the names of the object's properties must conform to the rules for PAN dictionaries. Notably, JSON property names starting with a number are not allowed.

### 3.1.36 json_encode

#### Name

json_encode – convert a PAN data structure to a JSON-encoded string

#### Synopsis

string **json_encode** (element *arg*)

#### Description

The `json_encode` function returns the JSON representation of the argument. It is currently not possible to control the JSON formatting.

Trying to encode `undef` either directly or embedded into a resource will fail. `null` values in lists will appear in the output, but dictionary keys with the value being `null` will not be serialized.

### 3.1.37 length

#### Name

length – returns size of a string or resource

#### Synopsis

long **length** (string *str*, long *length*, resource *res*)

#### Description

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

### 3.1.38 list

#### Name

list – create a new list consisting of the function arguments

#### Synopsis

list **list** (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');
```

### 3.1.39 long_to_ip4

#### Name

long_to_ip4 – converts a long into an IP address in dotted format

#### Synopsis

string **long_to_ip4** (long *ip*)

#### Description

The `long_to_ip4` function converts an IP address represented as a long into a string with numbers and dots, like `inet_ntoa` does in the C standard library.

```
"/ipaddr" = long_to_ip4(0x01020304); # 1.2.3.4
```

### 3.1.40 match

#### Name

match – checks if a regular expression matches a string

#### Synopsis

boolean **match** (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)$');
```

### 3.1.41 matches

#### Name

matches – returns captured substrings matching a regular expression

#### Synopsis

string[] **matches** (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);
};
```

### 3.1.42 merge

#### Name

merge – combine two resources into a single one

#### Synopsis

resource **merge** (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 dicts. If more than one dict 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'));
```

### 3.1.43 next

#### Name

next – increment iterator over a resource

#### Synopsis

boolean **next** (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.

### 3.1.44 path_exists

#### Name

path_exists – determines if a path exists

#### Synopsis

boolean **path_exists** (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.

### 3.1.45 prepend

#### Name

prepend – adds a value to the beginning of a list

#### Synopsis

list **prepend** (element *value*)

list **prepend** (list *target*, element *value*)

list **prepend** (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);
};
```

## 3.1.46 replace

### Name

replace – replace all occurrences of a regular expression

### Synopsis

string **replace** (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.

## 3.1.47 return

### Name

return – exit DML block with given value

### Synopsis

element **return** (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));
};
```

## 3.1.48 splice

### Name

splice – insert string or list into another

### Synopsis

string **splice** (string *str*, long *start*, long *length*, string *repl*)

list **splice** (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.
```

## 3.1.49 split

### Name

split – split a string using a regular expression

### Synopsis

string[] **split** (string *regex*, string *target*)

string[] **split** (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.

## 3.1.50 substitute

### Name

substitute – substitute named values in string template

### Synopsis

string **substitute** (string *template*) string **substitute** (string *template*, dict *substitutions*)

#### Description

The `substitute` function will replace all named values in the template, delimited like '${myvar}', with associated values. If only one argument is given, then the values will be looked up in the local and global variable definitions. If two arguments are given, then the lookup will be done in the explicit dict provided; this form will *not* use local or global variable values.

```
variable vars = dict('freq', 3, 'msg', 'hello');

# produces string 'say hello 3 times'
'/result' = substitute('say ${msg} ${freq} times', vars);
```

The substitution allows for recursive references. If you need to have something like '${myvar}' literally in the string, then use '$${myvar}'. If the template references an undefined value, then an EvaluationException will be raised.

### 3.1.51 substr

#### Name

substr – extract a substring from a string

#### Synopsis

string **substr** (string *target*, long *start*)

string **substr** (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
```

### 3.1.52 to_boolean

#### Name

to_boolean – convert argument to a boolean value

#### Synopsis

boolean **to_boolean** (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.

### 3.1.53 to_double

#### Name

to_double – convert argument to a double value

#### Synopsis

double **to_double** (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.

### 3.1.54 to_long

#### Name

to_long – convert argument to a long value

#### Synopsis

long **to_long** (property *prop*)

long **to_long** (property *prop*, long *radix*)

#### 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 radix is supplied, then it will be used for the conversion. When using the radix, string values should not be prefixed with the radix. That is, use `to_long('ff', 16)` or `to_long('0xff')`.

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.

## 3.1.55 to_lowercase

### Name

to_lowercase – change all uppercase letters to lowercase

### Synopsis

string **to_lowercase** (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.

## 3.1.56 to_string

### Name

to_string – convert argument to a string value

### Synopsis

string **to_string** (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 dicts.

## 3.1.57 to_uppercase

### Name

to_uppercase – change all lowercase letters to uppercase

### Synopsis

string **to_uppercase** (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.

### 3.1.58 traceback

#### Name

traceback – print message and traceback to console

#### Synopsis

string **traceback** (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.

### 3.1.59 unescape

#### Name

unescape – replaces escaped characters with ASCII characters

#### Synopsis

string **unescape** (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.

### 3.1.60 value

#### Name

value – retrieve a value specified by a path

#### Synopsis

element **value** (string *path*) element **value** (string *path*, element *default*)

#### 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 and no (optional) default is provided. If a *default* element is defined as second argument, and either there is no element for the given path or the (current) element of the given path is *undef*, the default element is returned.

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

# /z will be (the default) 10
'/z' = value('/nopath', 10);

# /v will be (the default) 5
'/u' = undef;
'/v' = value('/u', 5);
```

## 3.2 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.

### 3.2.1 panc

#### Name

panc – compile pan language templates

#### Synopsis

panc [`--no-debug` | `--debug`] [`--debug-ns-include regex`] [`--debug-ns-exclude regex`] [`--initial-data dict-dml`] [`--include-path path`] [`--output-dir dir`] [`--formats formats`] [`--java-opts java-options`] [`--max-iteration limit`] [`--max-recursion limit`] [`--nthread number`] [`--no-disable-escaping` | `--disable-escaping`] [`--logging string`] [`--log-file file`] [`--warnings flag`] [`-v` | `--no-verbose` | `--verbose`] [`-h` | `--no-help` | `--help`] [`--version`] [template ...]

#### Description

The `panc` command will compile a collection of pan language templates into a set of machine configuration files. This command, with its reorganized and simplified options, replaces the older `panc` command.

**`--no-debug, --debug`** Enable or disable all debugging. By default, debugging is turned off.

**`--debug-ns-include=`** 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-ns-exclude=`** 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.

**--initial-data=** A DML expression that evaluates to an dict. This value will be used as the starting dict for all object templates. This is a convenient mechanism for injecting build numbers and other metadata in the profiles.

**--include-path=** Defines the source directories to search when looking for templates. The value must be a list of absolute directories delimited by the platform's path separator. If this is not specified, the current working directory is used.

**--output-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.

**--formats=** A comma separated list of desired output formats. Allowed values are "pan", "pan.gz", "xml", "xml.gz", "json", "json.gz", "txt", "dep", "dep.gz" and "dot". The default is value is "pan,dep".

**--java-opts=** 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.

**--max-iteration=** Set the limit on the maximum number of permitted loop iterations. This is used to avoid infinite loops. The default value is 5000.

**--max-recursion=** Set the limit on the maximum number of permitted recursions. The default value is 10.

**--nthread=** The number of threads to use for profile processing. The default value of zero will use a number equal to the number of CPU cores on the machine.

**--no-disable-escaping, --disable-escaping** Enable or disable the escaping of path elements. The default value is to enable the escaping of path elements.

**--logging=** Enable compiler logging; possible values are "all", "none", "include", "call", "task", and "memory". A log file must be specified with the `--log-file` option to capture the logging information.

**--log-file=** Set the name of the file to use to store logging information.

**--warnings=** Possible values are "on", "off", and "fatal". The last value will turn all warnings into fatal errors.

**-v, --no-verbose, --verbose** At the end of a compilation, print run statistics including the numbers of files processed, total time, and memory used. The default is not to print these values.

**-h, --no-help, --help** Print a short summary of command usage if requested. No other processing is done if this option is given.

**--version** Prints pan compiler version.

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.

### 3.2.2 panc-annotations

#### Name

panc-annotations – process annotations in pan configuration files

#### Synopsis

panc-annotations [`--base-dir base-directory`] [`--output-dir dir`] [`--java-opts jvm-options`] [`-v` | `--no-verbose` | `--verbose`] [`-h` | `--no-help` | `--help`] [`--version`] [template . . . ]

#### Description

The `panc-annotations` command will process the annotations contains within pan configuration files within the given base directory.

**--base-dir=** Defines a base directory containing all pan configuration files to process. The default is value is the current working directory.

**--output-dir=** Set where the annotation files will be written. If this option is not specified, then the current working directory is used by default.

**--java-opts=** 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.

**-v, --no-verbose, --verbose** At the end of a compilation, print run statistics including the numbers of files processed, total time, and memory used. The default is not to print these values.

**-h, --no-help, --help** Print a short summary of command usage if requested. No other processing is done if this option is given.

**--version** Prints pan compiler version.

The `panc-annotations` command is just a wrapper script around the `java` command to simplify setting various options.

### 3.2.3 panc-build-stats.pl

#### 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)
```

## 3.2.4 panc-call-tree.pl

### 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 can be used to visualize graphs written in dot format. Hypergraph 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.

## 3.2.5 panc-compile-stats.pl

### 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)
```

### 3.2.6 panc-memory.pl

#### 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)
```

### 3.2.7 panc-profiling.pl

#### 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.

### 3.2.8 panc-threads.pl

#### 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)
```

## 3.3 Getting the Compiler

You will need to download and install the pan language compiler in order to use it. This chapter explains where to obtain the compiler and how to install it.

### 3.3.1 Binary Distributions

Current binary packages (v10.1 and later) are available from the GitHub repository in a variety of formats.

```
https://github.com/quattor/pan/releases
```

Older releases (previous to 10.1) are available from SourceForge:

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

### 3.3.2 Source

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

```
git clone git://github.com/quattor/pan.git
```

This provides a *read-only* copy of the pan repository. Patches to the compiler can be provided via GitHub pull requests.

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 1.6.0 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 3; the build should work for any Maven version 2.2.1 or later.

### 3.3.3 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 1.6.0 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.

## 3.4 Running the Compiler

To facilitate the use of the pan configuration language compiler in different contexts, several mechanisms for running the compiler are supported, ranging from direct invocation from the command line to use within build frameworks like ant and maven.

The performance of the compiler can vary significantly depending on how the compiler is invoked and on what options are used. Some general points to keep in mind are:

- For large builds, try to start the underlying Java Virtual Machine (JVM) only once. That is, avoid the command line interface and instead use one of the build framework integrations.
- The pan compiler can be memory-intensive to medium to large-scale builds. Use the verbose output to see the allocated and used heap space. Increase the allocated memory for the JVM if the used memory exceeds about 80% of the total.
- Other JVM optimizations and options can improve performance. Check out what options are available with your Java implementation and experiment with those options.

The following sections provide details on the supported mechanisms for invoking the pan configuration language compiler.

### 3.4.1 Command Line

The compiler can be invoked from the command line by using `panc`. This is a script, which works in Unix-like environments, that starts a Java Virtual Machine and invokes the compiler.

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

### 3.4.2 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. The options are the same as for the `panc` command, except that the java options parameter is not supported.

### 3.4.3 Maven

The pan compiler release contains a simple maven plug-in that will perform a pan syntax check and build 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.

> **Warning:** The options of the plug-in have changed from the previous version. They mirror those of the `panc` script. Details for the options are given below.

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=9.3

...

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 plug-in. You can also obtain more detailed help via the maven help system

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

The following tables show the available parameters for the PanBuild and PanCheckSyntax mojos.

Table: PanBuild Mojo Parameters

| Parameter | Description | Required |
|---|---|---|
| sourceDirectory | Location of pan language sources. | No. Default value: '${basedir}/src/main/pan' |
| verbose | Whether to include a summary of the compilation, including number of profiles compiled and overall memory utilization. | No. Default value: false |
| warnings | Sets whether warnings are printed and whether they are treated as fatal errors. Allowed values are 'on', 'off', and 'fatal'. | No. Default value: 'on' |

Table: PanCheckSyntax Mojo Parameters

### 3.4.4 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 task definition element 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 four tasks defined:

**panc** Provides all of the functionality available through the compiler.

**panc-check-syntax** Checks only the syntax of the pan source files. This is the recommended way of doing a syntax check.

**panc-annotations** Processes panc annotations found in the templates and produces XML files with the resulting content.

**panc-version** Displays the pan compiler version.

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

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

  <panc-annotations OPTION="VALUE" >
    <fileset dir="${basedir}/profiles" casesensitive="yes" includes="*.pan" />
  </panc-annotations>

</target>
```

where `OPTION="VALUE"` is replaced with valid options (attributes) for the pan compiler ant tasks. The following tables describe all of the attributes supported by the these tasks (task `panc-version` accepts no option).

Table: Attributes for Ant Task `panc`

| Option | Description | Required |
|---|---|---|
| warnings | Sets whether warnings are printed and whether they are treated as fatal errors. Allowed values are 'on', 'off', and 'fatal'. | No. Default value: 'on' |
| verbose | Whether to include a summary of the compilation, including number of profiles compiled and overall memory utilization. | No. Default value: false |

Table: Attributes for Ant Task `panc-check-syntax`

| Option | Description | Required |
|---|---|---|
| baseDir | Base directory used to locate the templates if their names is a relative path and to build the relative path used to create output file if the path is absolute. | Yes. |
| outputDir | Parent directory used to create output XML files. The output file name is built by appending the template relative path to this directory. | Yes. |
| verbose | If true, displays statistics after processing the annotations. | No. Default: false. |

Table: Attributes for Ant Task panc-annotations

#### Nested Elements

Some of the configuration options are specified via nested elements. The panc task supports all of these; the panc-check-syntax and panc-annotations task only supports the fileset nested 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 ="-Xmx1024M"
```

or

```
setenv  "-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.)

### 3.4.5 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.

### 3.4.6 Displaying the compiler version

There are different ways of displaying the pan compiler version, depending on the invocation method.

| Invocation | Command |
|---|---|
| Java | java -jar /path/to/panc.jar |
| panc | panc –version |
| panc-annotations | panc-annotations –version |
| Ant | task panc-version |

Table: How to get panc version

# CHAPTER 4

## Search

- search