Pan Tutorial: A Whirlwind Tour of the Pan Language

C. Loomis (CNRS/LAL)

11th Quattor Workshop (CERN)
16-18 March 2011

Contents

- Basics
  - Download, install.
  - Declarative syntax
- Performance
- Idioms
- Advanced techniques

Raison d’être

- Purpose
  - Define (machine) configuration parameters
  - Subject to (user-defined) validation criteria
- Goals
  - Simple, human-friendly syntax
  - Same language for parameters, validation, etc.
  - Easy reuse & sharing of configuration information

Place in the Ecosystem

- Workflow is nearly identical to that for standard software development. Between VCS and “actuators”:
  - Version control system: SCDB, CDB, …
  - Quattor NCM subsystem: configuration components, …
- In principle, could be used with any VCS and used for any type of configuration.

Download & Installation

- To follow exercises:
  - Download latest panc tarball from SF
  - Requires version of Java JDK or JRE 1.6
  - Setup environment (PATH=...) for panc
- SourceForge links
  - Use v8.4.7
  - http://sourceforge.net/projects/quattor/files/panc/

Hello World

```
#
# hello_world.pan
#
object template hello_world;

‘/message’ = ‘Hello World!’;
```

```
$ panc hello_world.pan
$ ls hello_world.*

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

Declarative Language

- Primary statement is an assignment!

```
'/my/path' = 47;
```

- Define tree of configuration parameters:
  - Syntax similar to unix file system
  - Looks very much like proc file system on linux
- Other statements:
  - Template declaration
  - (Global) variable, type, or function definitions
  - Binding statement: types applied to path

Declarative Language (2)

- Feel yourself missing procedural flow control in templates?
  - Very likely an opportunity to capture and reuse some configuration into separate templates.
  - Or something that is better done in the perl code of a configuration module.

Statements

Statement	Purpose
‘/path’ = ‘OK’;	unconditionally assign the value to the given absolute or relative path
‘/path’ ?= ‘OK’;	conditionally assign the value to the given absolute or relative path
include { ‘other_template’ };	include and execute the statements in the other template; if name if undef or null, nothing is done
variable X = ‘OK’;	create global variable X with the value ‘OK’
variable X ?= ‘OK’;	conditionally set the variable X to the value ‘OK’
type x = string;	define type x to be a string
function x = 42;	define function x that always returns 42
bind ‘/path’ = x;	bind type definition x to the path ‘/path’
prefix ‘/path’;	sets the path prefix to ‘/path’ for any subsequent relative assignment statements

Types of Templates

Modifier	Name	Purpose
object	object template	signals that a profile (*.xml file) should be generated
<none>	ordinary template	contains any type of statement for inclusion by other templates
unique	unique template	like an ordinary template but will be executed only once for each profile
declaration	declaration template	may only contain variable, type, and function definitions; only executed once for each profile
structure	structure template	contains only relative assignment statements; included via the create() function

Batch System

- Use example of a simple batch system to show major features of pan language.
- Batch system (or cluster) has a “head node” that accepts job requests and farms them out to a number of worker nodes for execution.
- Server: has nodes (each node participates in queues, has capabilities), has queues (each queue has CPU limit)
- Worker: references server, enabled/disabled

Batch1

- Simple example showing how to declare the configuration for server and 1 worker.

- Not very extensible organization:
  - All of the templates at root level.
  - Copy/paste duplication with workers.

Batch2

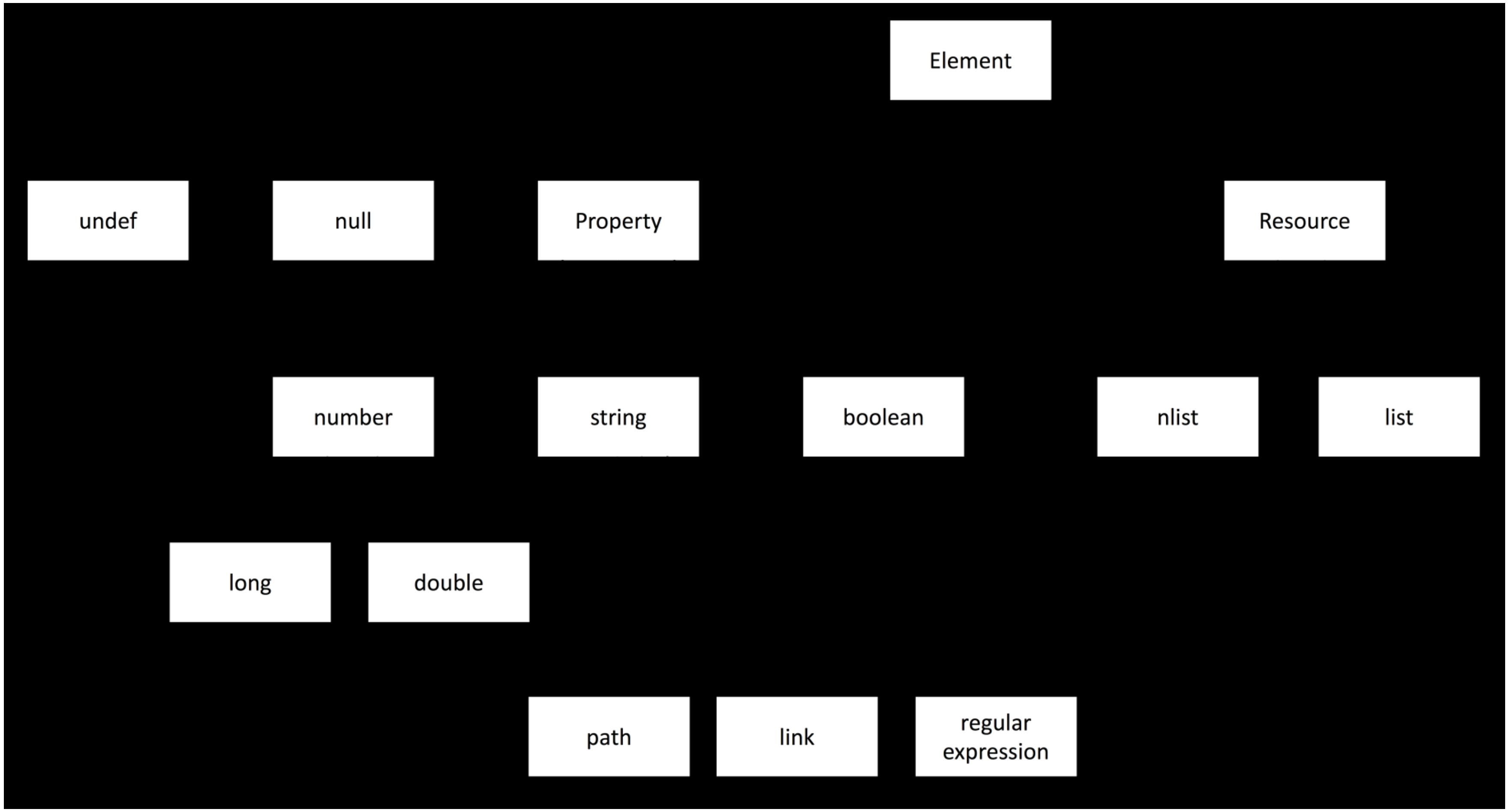
- Split service configuration from node declarations.
- Use namespaces for service and node declarations.
- Can make more workers with less duplication.

- Doesn't provide protection against bad values in the configuration.

Batch3

- Add type declarations:
  - Boolean, longs, etc.
- Often default values make sense and would like to define values only if different than the default.

Type Hierarchy



Batch4

- Provide default values:
  - Can provide resources as well as properties!
  - Defaults only provided if the parent exists.

- Would like to provide consistency checks between values and between node declarations.

Batch5

- Cross-value/cross-machine validation:
  - Ensure that listed queues actually exist.
  - Ensure that server and workers all know about each other.

- Templates often modify multiple parameters in the same part of the configuration tree.

Batch6

- Use of the 'prefix' statement:
  - Pseudo-statement: only affects containing template
  - Best practice: one prefix at beginning of template

Common Problems

- Last statement executed provides the value of a DML block.
  - All DML statements provide a value, even flow control statements like if/else, foreach, while, etc.

```
'/path' = if (false) 'MY VALUE';  # returns undef
```

  - Use care when assigning to resources in DML block.

```
'/path' = {
  m['child'] = 3;
};                     # Value is 3, not nlist!
```

Performance

- Be explicit with paths, push as much information to left of assignments as possible.

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

# More legible and faster.
'/path/a' = 1;
'/path/b' = 2;
```

- Invoking compiler:
  - Avoid panc script if possible. You pay the startup costs of the JVM every time it is invoked.
  - Ant/maven are more effective and provide dependency management as well.

Performance

- Use escaped literal syntax:

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

# More legible and faster.
'/path/{a/b}' = 1;
```

- Always use a built-in function instead of a function defined in pan!
  - Especially important for append(), prepend()
  - Look at to_uppercase(), to_lowercase(), etc.

Performance

- Avoid SELF if possible!
  - Avoid incremental builds of lists, rearranging the configuration, if possible.
  - Always (!!) use SELF directly in any DML block. Do NOT copy to a local variable!

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

'/path' = {
  copy = SELF;      # Deep copy of SELF!
  copy['c'] = 3;
  SELF;             # Added value is LOST.
};
```

Idioms

- Default variables for modifying configuration.

```
variable MY_SERVICE_CONFIG ?= null;   # or undef
include { if_exists(MY_SERVICE_CONFIG) };

variable ADD_NFS_MOUNT ?= null;
'/mounts' = {
  if (ADD_NFS_MOUNT) {
    '/var/log …';
  } else {
    null;
  };
};
```

Idioms

- null is useful for tri-state variables or sentinel values:

```
variable X = true; # or false or null

'/path' = X;   # completely unset if null
```

- Use file_contents() and format() for simple configuration files.

```
variable X = file_contents('my_cfg_file');

'/path' = format(X, 10, 20, 'USER');
```

Advanced Techniques

- Annotations
- Logging/debugging:
  - Can generate dependency information
  - Use verbose for performance stats
  - Use memory, call, … logging for detailed analysis
  - Use debug() function for detailed information
  - Use traceback() to find problem location

Documentation

- Please read the documentation!
  - Compiler and language manuals (pan-book).
  - README often has useful information!