The L3 Configuration Language
Draft Manual

Paul Anderson <dcspaul@ed.ac.uk>
School of Informatics
University of Edinburgh

This document describes the language and prototype compiler for the L3 configuration language. Sections marked (⊿) describe more advanced features which may refer to later sections the document and can be omitted on a first reading.

1 The L3 Language

1.1 Comments

The language supports Java-style comments:

```plaintext
/*
 * block comment
 */

// single line comment
```

1.2 Resources

A resource is a labelled value (§1.4):

```plaintext
a: 45
X_21 : "hello"
```

If the the label and value are separated by a colon, then the resource is published to the output. If they are separated by an equals sign, then the resource is not published to the output, but may still be referenced from elsewhere in the configuration (§1.7):

```plaintext
published_res : "stuff"
unpublished_res = 25
```

1.3 Labels

A simple label is a string of alphanumeric or underscore characters:

```plaintext
a_label: 45
```

(⊿) Labels may also be arbitrary expressions enclosed in brackets. The expression must evaluate to a string:

```plaintext
("a"++"label"): 45
```

(⊿) Care is required when using evaluated labels which contain references (§1.7), since the labels are evaluated in the context of the enclosing block (not the current block) and the results may be confusing:

```plaintext
x = "this"
y: { x = "not this" ($x): 1 }
⇓
y: {
  ( "this" ): 1
}
```

1.4 Values

The primitive types of the language are the usual boolean, string, and number. Numbers may be integers or floating point. Resources whose value is none (§1.14) or undef (§1.15) are not published in the output.

```plaintext
a: true
b: "hundred acre wood"
c: 23
d: 56.75
e: none
f: undef
```

Strings may include the usual quoting and escape characters, as well as "here documents":

```plaintext
a: "a multi-line
string"
b: >>>
  a string which
  includes several lines
  of verbatim text
<<<
```

A value may also be an expression involving any of the following:

- An block (table 1.5)
- An operator (table 1)
- A conditional (§1.6)
- A reference (§1.7)
- A selection (§1.8)
- A list (§1.9)
- A composition (§1.10)
- A choice (§1.11)
- A specialisation (§1.13)
- none (§1.14)
- undef (§1.15)
- A function call (§1.16)
string= "mail"
fun= fail "bad name"
cond: if ($string=="mail")
    then 22 else $fun
block: { ref: $string, port: $cond }
comp: $block <+> { port: 46 }
spec: $block +> { port: 46 }
sel: $spec . port

Table 1: Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-=</td>
<td>unary minus</td>
</tr>
<tr>
<td>a * b</td>
<td>multiply</td>
</tr>
<tr>
<td>a / b</td>
<td>divide</td>
</tr>
<tr>
<td>a + b</td>
<td>add</td>
</tr>
<tr>
<td>a - b</td>
<td>subtract</td>
</tr>
<tr>
<td>a ++ b</td>
<td>string concatenation</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>greater than</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>less than</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>not less than</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>not greater than</td>
</tr>
<tr>
<td>a == b</td>
<td>equal</td>
</tr>
<tr>
<td>a /= b</td>
<td>not equal</td>
</tr>
<tr>
<td>!a</td>
<td>logical not</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>logical and</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

The operator table is in precedence order and brackets may be used in the normal way.

1.5 Blocks

A block is a list of resources enclosed in `{}`. The resources may be separated by whitespace or commas:

```
x: { a:1, b:2 }
y: { 
a:1
 b:2
}
```

Note that the order in which the elements appear in a block is not significant. Multiple labels can be used as a shorthand for defining nested blocks:

```
long: { a = { b: "stuff" } }
short:a=b:RD "stuff"
```

(>) If the same label appears more than once in a block, then the values corresponding to each occurrence are composed (§1.10). For example, the following are equivalent:

```
x: { b:1, c:2, b:3 }
y: { b:1 <+> 3, c:2 }
```

Values appearing without labels are raised and composed with the current block. For example, the following are equivalent:

```
x: { b:1 { c:2 } }
y: { b:1 } <+> { c:2 }
z: { b:1 c:2 }
```

This is particularly useful for import statements (§1.16.3) and conditionals (§1.6):

```
import "ports"
port: $server_port
where ports.l3 contains:
server_port = 42
client_port = 43
```

1.6 Conditionals

Conditionals are expressions - they always return a value. The condition must evaluate to a boolean value. The branches are evaluated lazily. If the else clause is omitted and the condition is false, then the result is the undefined value:

```
cond = true
x: if ($cond) then 1 else fail "cond is false"
if (!$cond) then 2+"type error"
```

1.7 References

A value can be specified as a reference to any other resource:

```
a: "stuff"
b: $a
```

Default references are interpreted relative to the current block, and the compiler will search up the enclosing blocks for the “nearest” definition. In the following example, a.b.c.d has the value 2:

```
x: 1
a: {
x: 2
 b: { c: { d:$x } }
}
```

Absolute references are interpreted relative to the top block. In this example, a.b.c.d has the value 1:

```
1The order in which the resources will appear in the output is undefined, although the compiler will guarantee to produce the same output ordering for the same input.

Paul Anderson
If the target of a reference is itself a reference, then there is more than one possible way of interpreting the final result. For example: when \( c.x \) is evaluated in the following configuration, the \( \$a \) could be interpreted in the context of the \( b: \) (yielding 1), or in the context of the \( c: \) (yielding 2):

\[
\text{block: \{ a: 1 b: \$a c: \{ a:2 x: \$b \} \}}
\]

System administrators are familiar with a range of languages which interpret references in different ways. In order to experiment with the apparent usability of these, L3 can be globally configured to use different evaluation strategies (§3.1). The semantics can also be specified explicitly for individual references (§3.3). For example: \( \$[s]label \) (static) or \( \$[d]label \) (dynamic). However, this is not intended to be used as a “production” feature!

By default, the current L3 compiler will return a composition (§1.10) of values obtained from both of the above strategies. In the majority of practical cases, only one of these will be defined, so the resulting value will be the only valid interpretation. However, this expensive to implement, and complex cases can be extremely confusing, so this is unlikely to remain the default strategy.

### 1.8 Selections

The selection operator allows the value of one resource to be selected from a block. This is normally most useful when applied to a reference, although it can be applied to any expression which evaluates to a block:

\[
\text{a: \{ x:1 y:2 \} b: \$a.y c: \{ x:1 y:2 \} . x}
\]

As with references, there are two different interpretations of the semantics if the result of the selection contains a reference. The compiler compiles both of these in a similar way, and provides a similar notation for specifying an explicit semantics:

\[
\text{c: \{ a:2 x: \{ a:1 b: \$a \} . b \}}
\]

As with labels, the selector itself may be specified as an arbitrary expression (in brackets), although with the same caveats.

\[
\text{x: \{ foobar: 3 \} . \("foo"++"bar"\)}
\]

### 1.9 Lists

L3 lists are ordered lists of values. All elements of the list must have the same type. The concatenation operator is overloaded to operate on lists as well as strings:

\[
\text{a: [ 42, 6, 103 ] b: [ "foo", "bar" ] c: \$a ++ [ 5, 6 ]}
\]

List elements may be given labels. These are discarded in the output, but they can be used to control the order in which elements appear when two lists are composed (§1.10).

### 1.10 Compositions

The composition operation (\(<++>\)) allows a configuration to be assembled from multiple partial specifications. The composition operation is commutative: configuration blocks (usually from different authors) can be composed in an arbitrary order with the same result. For example:

\[
\text{alice = \{ a:1, b:2 \} bob = \{ c: 3 \} result: \$alice <++> \$bob} \downarrow \text{result: \{ a: 1 b: 2 c: 3 \}}
\]

When two value are composed, the result is published if either of the operands is published.

**Composing undefined values:** \( \text{undef} \) (§1.15) is an identity for composition:

\[
\text{x: 1 <++> \$nothing y: \$nothing <++> 2} \downarrow \text{x: 1 y: 2}
\]

**Composing primitive values:** Two primitive values which are equal compose to a single primitive value:

\[
\text{x: "foo" <++> "foo"} \downarrow \text{x: "foo"}
\]

Unequal primitive values of the same type compose to
The L3 Configuration Language

1.11 Choices

A “choice” allows more than one value to be explicitly specified as a candidate value for a resource:

```
x: 1 <|> 2 <|> 3
```

Choices also arise when two different values for the same resource are composed. This typically occurs (a) when there are two different possible targets for a reference (using two different reference interpretations); (b) when the same label occurs twice in one block; or (c) when two different blocks are composed but one of their resources has a common label with different values.

By default, the compiler will choose an arbitrary (but consistent) value from the available choices, but annotations (§1.12) can be used to guide the selection.

1.12 Annotations

Values may be annotated with tags and constraints which are used to guide the selection among alternative choices:

```
#alice > #bob
x: 1 #alice
x: 3 #bob
```

Tags are identifiers prefixed with a #. Constraints consist of two tags separated by < or > indicating a partial ordering constraint on the priority of the tags. Annotations immediately following a resource definition apply to that resource. Annotations at the start of a block apply to every resource in the block. Annotations are inherited from enclosing blocks, and propagated across expression evaluations.

Choices are resolved by selecting the highest priority value according to the partial order given by all of their accumulated constraints. An error occurs if there is no solution, and an arbitrary value is selected (with an optional warning) if more than one value is maximal.

Two tags are special: #final specifies a value that

---

2 Notice that this is a “deep” composition semantics, unlike SmartFrog.

3 i.e. the same compiler will produce the same output given the same input.
must be used for a particular resource - an error occurs if there is more than one (different) final value. #default specifies a value to be used only if there are no non-default values specified - multiple default values are resolved using their tag priorities in the normal way:

\[
\begin{align*}
a: & \{ x:1 \ #final \ y:1 \ #default \ } \\
\downarrow \\
a: & \{ \\
x: & 1 \\
y: & 2 \\
\}
\end{align*}
\]

Notice that this allows the authors of configuration components to specify how those components should compose. A “user” of these components can then compose them (in any order) without being aware of the underlying constraints. Example §2.1 shows how this may be used in practice.

### 1.13 Specialisation

Composition is a commutative operation – if there are multiple values for a particular resource, the choice of the final value is based on the annotations, not on the composition order. This means that complex configurations can be constructed by composing sub-configurations (aspects) without worrying about the order. For example, to create a machine which is both a “web server” and a “mail server”, we would expect both of these to be equivalent:

\[
\begin{align*}
\text{myserver: } & \$\text{webserver} \leftrightarrow \$\text{mailserver} \\
\text{myserver: } & \$\text{mailserver} \leftrightarrow \$\text{webserver}
\end{align*}
\]

However, this makes it difficult to explicitly override parameters. The following example would require the 35 to be marked as final — which may be too strong — or to depend on some annotations internal to the mailserver configuration:

\[
\begin{align*}
\text{myserver: } & \$\text{mailserver} \leftrightarrow \{ \text{port:35} \}
\end{align*}
\]

If we have a complete hierarchy of values which we would like to take priority, then this is much more difficult. For example, deptserver may contain a hierarchy of parameters which are appropriate for a specific department, and we may want to impose those onto the generic configuration for a mail server. This is not sufficient:

\[
\begin{align*}
\text{myserver: } & \$\text{mailserver} \leftrightarrow \$\text{deptserver}
\end{align*}
\]

In this case, we require something closer to the behaviour of other languages (such as SmartFrog) which use a standard instance inheritance semantics.

A specialisation (\(\leftrightarrow\)) is identical to a composition except that an additional constraint (and associated annotations) is added which prefers all elements of the second block to corresponding elements of the first. This provides the equivalent of conventional instance-inheritance semantics in a way which integrates well with the composition operations.

The composition operator binds more tightly than the specialisation operator, and brackets may be used when combining these:

\[
\begin{align*}
s: & \$\text{xserver} \leftrightarrow (\$\text{yserver} \leftrightarrow \{ \text{port:45} \})
\end{align*}
\]

### 1.14 None

Resources whose value evaluates to none are not included in the output when generating non-L3 formats. This can be used to remove resources from a prototype block. For example, JSON:

\[
\begin{align*}
x: & \{ a:1 \ b:2 \ } \leftrightarrow \{ a:\text{none} \} \\
\downarrow \\
\{ "x": \{ "b": 2 \} \}
\end{align*}
\]

Note that the composition of the none with other values is subject to the normal priority, as determined by the annotations.

### 1.15 Undef

Resources whose value evaluates to undef are not included in the output when generating non-L3 formats. This is similar to none, except that undef is the identity for composition, regardless of any priority specified by the annotations. undef most commonly appears implicitly as the result of references to a nonexistent resource, or other errors. For example:

\[
\begin{align*}
x: & \{ a:1 \ } \leftrightarrow \{ a: \$\text{missing} \ #\text{final} \} \\
\downarrow \\
\{ "x": \{ "a": 1 \} \}
\end{align*}
\]

Explicitly specifying undef as a resource value can be useful in certain circumstances, such as list composition (§1.9).

### 1.16 Functions

A number of built-in functions are supported:

#### 1.16.1 Map:

The map function applies an expression to each item of list. Within the expression, the variable \$_ represents the original value of the list item:

\[
\begin{align*}
x: & \text{map} (\$_+1) \ [ 3, 4, 5 ] \\
\downarrow \\
x: & [ 4, 5, 6 ]
\end{align*}
\]
This can also be applied to the values in a block:

```l3
x: map ($_.name) {
  a: { name="A", k=1 }
  b: { name="B", k=2 }
}

x: {
  a: "A"
  b: "B"
}
```

1.16.2 Fail: The `fail` function causes the compilation to fail with the given error message. The argument must be a string:

```l3
f: if ($x>0) then $x else fail "x <= 0"
```

1.16.3 Import: The `import` function allows the inclusion of source from other files. The argument must be a string:

```l3
x: import "somefile.l3"
```

The `import` expression usually appears without a label so that the resources defined at the top level of the included file are “raised” and inserted at the same level as the `import` statement itself:

```l3
import "somefile.l3"
```

The `.l3` extension may be omitted, and the compiler will search for the file using the path specified with the `-I` command line option.

If the `import` statement specifies a directory, then a new block is returned which includes a resource for every file in the directory which has a `.l3` extension. The name of each resource is the name of the corresponding file, and the value of the resource is an import statement to import the file.

Since the configuration is evaluated lazily, this provides a simple way to represent very large configurations: for example, a whole site configuration may be represented in a directory hierarchy and it is only necessary to import the top-level directory. Normal references can be used to reference any part of the configuration and (only) the necessary files and directories will be opened and parsed.

1.16.4 Trace: The `trace` function returns the value of its argument, displaying the value before and after evaluation:

```l3
x: trace 2+3
↓
[trace=op2] x: 2 + 3
[trace=int+] x: 5
x: 5
```

The `-DTrace` command line option can be used to display all of the steps in the evaluation of traced expression. The `-DEval` option enables tracing for the entire source file.

2 Examples

2.1 Priorities

The insignificance of statement ordering is an important feature of L3. This implies the need for some alternative mechanism to prioritise conflicting values. It is certainly possible to provide a very expressive mechanism using general constraints; however, this can quickly lead to error-prone configurations which are difficult to interpret and predict. L3 annotations are an attempt to provide sufficient expressiveness for most practical situations, while still remaining comparatively clear and simple. The following example shows how these might be used to control the composition of components from different authors:

The administrator of a service might create the following configuration file which can be imported by anyone wishing to use the service. Default values are provided for the path and permissions (readable by anyone) of the configuration file, and the suggested (conventional) directory for the configuration file is `/etc`:

```l3
configfile: {
  dir: "/etc" #convention
  path: $dir++"/foo" #default
  perms: 0644 #default
}
```

The following component is created by the security specialist: for a “secure” system, the permissions on all configuration files must be 0600 (readable only by the administrator). It is also suggested (but not mandatory) that the configuration files are located in `/secure`.

```l3
configfile: {
  dir: "/secure" #security
  perms: 0600 #final
}
```

The user can now create a “secure service” composition without understanding the details of how these interact, and without being concerned about the composition order:

---

4Tracing is also enabled for any closures returned by the evaluated expression, so additional tracing information may be displayed after the return value if these closures are evaluated later.

5See for example, ConfSolve.
import "inc/security"
import "inc/service"

configfile: {
    dir: "/etc"
    path: "/etc/foo"
    perms: 600
}

In this case the user has not specified any preference for the balance between "security" and "convention", so the system has (arbitrarily) chosen the location for the configuration file as /etc. However, the permissions have been set to the secure value of 0600 as expected. However, the user can explicitly specify a preference for convention over security:

#convention > #security
import "inc/security"
import "inc/service"

configfile: {
    dir: "/etc"
    path: "/etc/foo"
    perms: 600
}

Or visa-versa:

#security > #convention
import "inc/security"
import "inc/service"

configfile: {
    dir: "/secure"
    path: "/secure/foo"
    perms: 600
}

In general we would not encourage over-creative use of annotations, which can potentially lead to surprising results. For example, two resources which reference a common resource may evaluate differently if they are evaluated in contexts which have different annotations. In practice, the simple use of #default and #final are sufficient for many situations.

2.2 Map

host-info = { id:$name, ip:$addr }

hosts: {
    foo: $host +> {
        name: "foo.blob.com"
        addr: "1.2.3.4"
        other: "stuff"
    }

    bar: $host +> {
        name: "bar.blob.com"
        addr: "5.6.7.8"
        other: "stuff"
    }

    server: $info +> {
        name: "server"
        addr: "9.9.9.9"
        client_info: map $_.info $hosts
    }
}

This example shows how the map function can be used to collate specific information from a list of hosts ("clients") and make it available as a single resource (client_info) on some other host ("server"). This provides the same functionality as spanning maps in LCFG:

---

6The compiler would warn about this arbitrary choice if required.
The L3 Configuration Language (8)

3 The Compiler

The l3 command accepts a list of source files for compilation:

```
l3 foo.l3 bar.l3 ...
```

By default, the output files have the same name as the corresponding source files, with an extension which depends on the output format (specified by the -f command line option). The default is to generate files in L3 format with the extension .l3. The -o option can be used to specify a different directory for the output files, or to specify output to stdout (-o -).

3.1 Command line options

- `A r1,r2..`
  The semantics for absolute references (§3.3). Default: san,dan.

- `D ±d1,±d2..`
  Enable/disable debugging options (§3.4).

- `f format`
  The output format (§3.2). Default: l3.

- `I ±d1:±d2..`
  A (colon-separated) path list of directories to search for imported files.

- `L n`
  A limit on the number of evaluation steps. This is only effective when debugging evaluations (-DEval) and can be used to help debug non-terminating specifications. Default: 10000.

- `o dir`
  The output directory. The default is the same directory as the source file.

- `r l1:l2:..`
  The root path for the resource to be compiled. This allows for compilation of (only) the specified sub-tree of the whole configuration (which may be very large). The default is to compile the whole configuration.

- `R r1,r2..`
  The semantics for relative references (§3.3). Default: sur,dur.

- `S r1,r2..`
  The semantics for selections (§3.3). Default: s,d.

- `v`
  Verbose output.

- `X ±x1,±x2..`
  Enable/disable warnings and other options (§3.5).

3.2 Output Formats

The following output formats are supported:

- `l3` The default format. The output is in the same format as the input, but contains only primitive values, blocks and lists.

- `json` JSON format.

- `p1` Used for debugging. The same as l3 format, except that unpublished values are also displayed.

- `p0` Used for debugging. The parsed l3 format before evaluation.

3.3 Reference Semantics

The reference and selection semantics can be specified as a comma-separated list of evaluation strategies. The References (selections) will be evaluated using each of the specified strategies and the result will be a composition of all of these. Each strategy is a set of specifiers:

- `s|d` Static or dynamic (§1.7).
- `a|r` Absolute or relative (§1.7). Not valid for selections.
- `u|o` References search up the containing hierarchy, or refer only to the current block. Not valid for selections.

It is also possible to specify an explicit semantics for an individual reference (§[spec] label) or selection (§[spec] label) in the source file:

```
b: 0
x: { a: 1
  b: $a
  y: { a:2 z:{$[s,d,a]b } }
}
```

These options are intended to support an investigation of the various different semantics. There is no suggestion that this flexibility should be exposed in any “production” version of the language!
3.4 Debugging Options
The following debug options can be enabled (+) or disabled (-):

**Eval**  
Display evaluation of entire configuration (generates lots of output).

**Trace**  
Display complete evaluation of all traced expressions (§1.16.4). itemChoices Show all possible values for choices rather than evaluating (see example under §3.3). Since the choices are not evaluated, values which depend on their result in the output may not be evaluated correctly.

**Annotations**  
Show annotations on all debugging output (including p0 and p1 output).

**Provenance**  
Show provenance on all debugging output (including p0 and p1 output).

3.5 Warnings & Options
The following warnings/options can be enabled (+) or disabled (-):

**WarnChoices**  
Display a warning when the compiler makes an arbitrary selection from a choice.

4 Further Work
L3 is a work-in-progress. Current/future work includes:

- Usability evaluation with real configurations and administrators.
- Clarification and formalisation of the semantics, including references and annotation propagation.
- Rendering output formats for existing tools such as LCFG/SmartFrog.
- Compiler performance.
- Additional functions.