The L3 Configuration Language

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 (§.14):

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

If 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 (§.1.4) or undefined (§.1.5) are not published in the output.

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

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

```plaintext
a: "a multi-line\nstring"
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)
- undefined (§.1.15)
- A function call (§.1.16)

```plaintext
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
```
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:

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

Note that the order in which the elements appear in a block is not significant¹. Multiple labels can be used as a shorthand for defining nested blocks:

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

```plaintext
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:

```
Table 1: Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</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;&amp; b</td>
<td>logical and</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>
```

### 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:

```plaintext
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:

```plaintext
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 this example, `a.b.c.d` has the value 2:

```plaintext
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:

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

(☉) If the target of a reference is itself a reference, then there is more than one possible way of interpreting the final result. When `c.x` is evaluated in the following

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.
example, the $a$ could be interpreted in the context of
the b: (yielding 1), or in the context of the c: (yield-
ing 2):

```plaintext
block: {
  a: 1
  b: $a
  c: { a:2 x:$b }
}
```

The current implementation of the L3 compiler will
evaluate both of these (static and dynamic) interprer-
tations of the reference and return a composition (§1.10)
of the two values. In the majority of practical cases,
only one of these will be defined, so the resulting value
will be the only valid interpretation.

It may sometimes be useful for debugging to specify
an explicit reference semantics. The compiler supports
a notation for explicit static ($[s]label$) or dynamic
($[d]label$) references, as well as more complex
combinations. However, this should only be used for
debugging, and not as part of a live specification.

1.8 Selections

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

```
a: { x:1 y:2 }
b: $a.y
c: { x:1 y:2 }.x
```

(>) As with references, there are two different inter-
pretations of the semantics if the result of the selection
contains a reference. The compiler composes both of
these in a similar way, and provides a similar notation
for specifying an explicit semantics:

```
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.

```
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:

```
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 configura-
tion to be assembled from multiple partial specifica-
tions. The composition operation is commutative: con-
figuration blocks (usually from different authors) can
be composed in an arbitrary order with the same result.
For example:

```
alice = { a:1, b:2 }
bob = { c: 3 }
result: $alice <> $bob
  ⇓
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: $undef$ (§1.15) is
an identity for composition:

```
x: 1 <> $nothing
y: $nothing <> 2
  ⇓
x: 1
y: 2
```

Composing primitive values: Two primitive val-
ues which are equal compose to a single primitive
value:

```
x: "foo" <> "foo"
  ⇓
x: "foo"
```

Unequal primitive values of the same type compose to
a choice (§1.11):

```
x: 42 <> 43
  ⇓
x: 42 <|> 43
```

Composing blocks: Two blocks are composed by
taking the union of the resource labels and recursively
composing the values associated with each:

\[\text{Notice that this is a "deep" composition semantics, unlike SmartFrog.}\]
Composing lists: Two lists are composed by merging the elements. The complete order of the final list is undefined - however (a) the composition maintains the partial order defined by each of the arguments, (b) it is commutative, and (c) it is deterministic:

| a | 42, 6 |
| b | 5, 71 |
| x: | a <-> b |
| y: | 5b <-> a |
| z: | 5, 42, 6, 71 |
| y: | 5, 42, 6, 71 |

If any list elements are labelled, the labels will not appear in the output, but values with matching labels are composed in the usual way:

| a | 42, lab:6 |
| b | lab:5, 71 |
| x: | a <-> b |
| y: | 5b <-> a |
| z: | 42 |
| 6 <-> 5 |

This can be used in combination with the `undef` value to control the ordering of elements in the composition. For example - the list x will not itself contain a value corresponding to k when it is output (in JSON, for example). However, if it is composed with any list which does contain a value for k, then this value will appear in the result, before the 7:

| x: | k:undef, 7 |
| y: | k:71, 5 |
| z: | 5x <-> 5y |
| z: | 71, 5, 7 |

It is, of course, possible to use this to specify partial orders which have no solution. This is an error:

| x: | a:1, 2, a:3 |

Composing other types: Composing other combinations of types is illegal.

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 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:
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 §2.2 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:

```
myserver: $webserver <+> $mailserver
myserver: $mailserver <+> $webserver
```

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:

```
myserver: $mailserver <+> { port:35 }
```

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:

```
myserver: $mailserver <+> $deptserver
```

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 (+>) 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:

```
s: $xserver <+> ( $yserver +> { port:45 } )
```

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:

```
x: { a:1 b:2 } +> { a:none }
```

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 non-existent resource, or other errors. For example:

```
x: { a:1 } <+> { a:$missing #final }
```

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:

```
x: map ($_+1) [ 3,4,5 ]
```

This can also be applied to the values in a block:
The L3 Configuration Language

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
```

```l3
[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 References

Defining the reference (and selection) behaviour in a way which matches reasonable expectations is not straightforward. The following example shows a realistic configuration fragment which illustrates the problem well: this starts with the definition of an abstract service component, and the intention is to allow client and server machines to reference the corresponding sub-components, and hence guarantee that the client and server ports will always match (because they both refer to the same `service.port` resource):

```l3
service = {
  port: 25 #default
  client: { port: $port } // ~> { ... } 
  server: { port: $port } // ~> { ... } 
}
myserver = $service +> { port: 26 }
machineA: $myserver.client // ~> { ... }
machineB: $myserver.server // ~> { ... }
```

However, there is no single interpretation of the reference semantics which yields the expected result: if the references are evaluated immediately, in the context in which they appear (`static`), then the client and server ports are always bound to the value 25:

```l3
machineA: { port: 25 }
machineB: { port: 25 }
```

If the evaluation is deferred (`dynamic`), then the client and server on will have an undefined port, since there is no target for the reference by the time it is evaluated:

```l3
machineA: { port: undef }
machineB: { port: undef }
```

To obtain the desired result, the references must be evaluated in a different way in different situations. It would be possible to provide separate notations and allow the user to specify the appropriate semantics for every individual reference (and selection), but this requires the user to make a careful and error-prone de-
cision in every case\(^5\).

The default L3 behaviour is to compose the two interpretations. This usually produces the expected result automatically, since the “wrong” values in each case are either undefined or annotated as defaults:

```
import "inc/security"
import "inc/service"

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

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 Priorities

The insignificance of statement ordering is an important feature of L3\(^6\)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 very 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:

```
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.

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

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:

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

\(^5\)LCFG provides this feature which is rarely used in practice.

\(^6\)See for example, ConfSolve.

### 2.3 Map

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:

```
map {
  dir: "/secure" #security
  path: "/secure/info"
  perms: 0600
}
```

\(^7\)The compiler would warn about this arbitrary choice if required.
The L3 Configuration Language

The L3 Configuration Language (8)

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
  }
}

hosts: {
  bar: {
    addr: "5.6.7.8"
    name: "bar.blob.com"
    other: "stuff"
  }
  foo: {
    addr: "1.2.3.4"
    name: "foo.blob.com"
    other: "stuff"
  }
  server: {
    addr: "9.9.9.9"
    client_info: {
      bar: {
        id: "bar.blob.com"
        ip: "5.6.7.8"
      }
      foo: {
        id: "foo.blob.com"
        ip: "1.2.3.4"
      }
      server: undef
    }
    name: "server"
  }
}

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 -out.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: 13.
-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.
pl
   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.

Paul Anderson
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 }
}
```

```
|↓|
```

```
b: 0
x: {
  a: 1
  b: 1 <|> 1
  y: {
    a: 2
    z: 1 <|> 1 <|> 2 <|> 2 <|> 0
  }
}
```

These should be used only to experiment and debug configurations - they should not be used in “production” configurations.

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:

- Clarification of the semantics of annotation propagation.
- Rendering output formats for existing tools such as LCFG/SmartFrog.
- Compiler performance.
- Additional functions.
- Formalisation of the evaluation semantics.