CF2

A Configuration Compiler

Paul Anderson <dcspaul@nine3.org>
nine3.org

1 Background

CF2 is a configuration compiler: it evaluates a textual configuration description and returns a simple data structure with key-value pairs for the configuration parameters.

CF2 is a natural successor to LCFG [1, 2]. The LCFG language has a very simple model which has proven to be particularly effective, in contrast to some other languages which have developed much more complex semantics (for example [6]). There is good evidence that even simple configuration languages are often poorly understood [8], difficult to use well, and a potential source of errors. CF2 attempts to retain the simplicity and usability of LCFG's declarative specifications while addressing some of the language's shortcomings. It deliberately avoids adding potentially "powerful" features (for example, the inference of ConfSolve [7]) where they may make the language more difficult to use.

While the basic LCFG model is sound, the implementation is old and extremely informal. It suffers from a number of problems, including:

- There is no formal syntax for the LCFG "language" and the compiler relies on the C preprocessor to handle constructions such as conditionals and file inclusion. This causes a number syntactic quirks (LCFG and C have a different syntax) and makes it impossible for other programs to parse LCFG specifications.
- All file inclusion happens at the pre-processing stage, so there is no way of restricting the scope of a particular file for security purposes.
- The LCFG compiler is intended to to operate in a centralised environment, compiling an entire site in one process. This can be extremely slow when large numbers of machines are involved.
- Because the configurations are completely precompiled on a server, there is no good way of incorporating parameters which are know only to the client at runtime (LCFG *contexts* are a workaround for this).

These are some of the issues which CF2 attempts to address.

2 Introduction

CF2 is a fundamentally a data description language – it is closer to a data specification such as XML than a programming language such as Perl, or a functional language. It consists primarily of a set of key-value pairs (known as *resources*) which define the configuration parameters. Records and lists are supported as well as scalar values, and a number of functions are provided to transform individual resource values, but it does not support arbitrary computation. It is the responsibility of the (separate) deployment engine to interpret these resources and implement the configuration. Section 3 describes the features available for defining the resource specifications.

Mutation: The distinguishing feature of CF2 (inherited from LCFG) is what LCFG calls *mutation*: The code for a typical program is created by combining code from many different modules, usually with different authors. Similarly, the resources in a large configuration refer to different *aspects* of the configuration which are the responsibility of different people. Many of the resources in these different aspects will be disjoint and the overall configuration can be created simply by merging the sets of resources (a union). In some cases however, the final value of a resource depends in a more complex way on the values specified for the different aspects – for example: the overall disk space required is the sum of the disk space required by each aspect; and the overall set of packages to be installed is the union of the sets of packages required by the different aspects. CF2 mutation functions (section 4) provide a way of specifying how multiple values for a resource are composed to create an overall value.

2.1 Language Formalities

Syntax: CF2 has a formal (LL(1)) grammar (appendix A). The reference parser uses functional combinators generated directly from the BNF by the PFC parser tool [4]. The parser modules can be used independently by a Perl program to produce an AST from CF2 source. This makes it easy to create other applications which analyse and process CF2 specifications.

Semantics: There is no formal semantics for CF2. However, the evaluator follows an operational semantics and the decision to avoid complexities in the language means that this is mostly straightforward. Some potentially useful features have been deliberately excluded to avoid over-complexity – for example rela-

tive references, which are particularly difficult to interpret in a meaningful way when combined with mutation (see [3]). The one exception is the *inline* use of blocks, conditionals, and functions – this has been included because it is particularly useful, but it has a difficult semantics and the implementation will reject some inputs which are technically valid.

2.2 Using the CF2 Compiler

Installation: CF2 is distributed as part of the PTools package. Installation simply requires downloading and unpacking the tar file from the PTools web page[1]. See the PTools documentation [5] for more details.

From Perl: The `CF2::Config` module can be used to compile the configuration from within a Perl program and make the resources available as autoloaded methods:

```
use CF2::Config;
my $cfg = new CF2::Config('./foo')
  ? $@ : die $@;
print $cfg->SomeResource
```

Perl programs using the PTools framework [5] need only declare a `Config` command line option (usually with `-c`) to specify the name of the configuration. The global method `Conf` can then be used to compile the configuration on demand and access the resource values.

```
print Conf->SomeResource
```

CF2 lists appear in the compiled configuration as Perl lists, and the blocks appear as `CF2::Config` objects which can be accessed using the autoloaded methods or hash references:

```
print Conf->Users->[0]
print Conf->UIDs->john
print Conf->UIDs->{'jane'}
```

From The Command Line: CF2 specifications can be compiled from the command line:

```
➔ ptools cf2 compile ./foo.cf2
```

By default, a fully evaluated top-level configuration is written to the standard output in CF2 format.

The `-o` option can be used (multiple times if required) to output different subtrees of the configuration to different files, in different formats. For example:

```
➔ ptools cf2 compile ./foo.cf2 \
        -o foo.json:a.b
```

[1] https://www.nine3.org/dcspaul/ptools.html

will write the value of the resource `a.b` to the file `foo.json` in JSON format. The supported formats (file extensions) are `cf2`, `json`, and `pl` (perl). The default format is `cf2` and the filename '`-`' refers to the standard output. For example:

```
➔ ptools cf2 compile ./foo.cf2 \
        -o -.json
```

will write the top-level configuration to the standard output in JSON format.

The `-Deval` option is useful to display a trace of the evaluation:

```
➔ ptools cf2 compile ./foo.cf2 -Deval
```

3 Resource Reference

A *resource* represents one configuration parameter. It consists of a *name* and an associated *value*. A basic resource definition uses the => (*assign*) operator to declare the resource and assign a value to it:

```
Directory => '/home/foo'
UID => 37
```

The resource assignments may be separated by newlines or commas. Comments can be included using `//`:

```
// this is a comment
// comments are ignored
Size => 123 // disk size
```

If a resource is declared more than once in the same file, the declarations must have the same value – this is not normally useful and the command line option `-Wdup` can be used to warn about duplicate assignments.

Resource names are standard identifiers, possibly including unicode characters. Values are expressions involving the following terms:

3.1 Scalars

Four types of simple scalar value are supported:

Text Values (Strings & Literals): *Literals* are enclosed in single quotes. Simple alphanumeric literals may be specified without quotes.

Strings are enclosed in double quotes. References and selectors (sections 3.5 and 3.6) may be interpolated into strings using `${...}`.

In both cases, the delimiters (and the backslash character itself) can be escaped with the backslash character, and `\n` and `\t` can be used to include newlines and

tabs respectively.

```
String1 => "two\nlines"
String2 => "${Literal2} is simple"
Literal1 => 'a literal'
Literal2 => simple123
```

The literals `true` and `false` represent boolean values (see below) so these must always be quoted if they are intended as text values.

Numbers: Numbers are straightforward integer or decimal values with an optional preceding minus sign:

```
OneTwoThree => 123
Decimal => -45.67
```

Notice that numbers are treated differently from the equivalent text values in some instances: for example:

```
X => '1' > '02'
Y => 1 > 02
```

`X` will be `true` because of the lexical comparison, and `Y` will be `false` because of the numeric comparison:

```
X => true
Y => false
```

Booleans: Booleans are represented as `true` or `false`. Again, these are treated differently from the corresponding text values in certain circumstances:

```
X => true || false
Y => 'true' || 'false'
```

`X` will be `true`, but the second line will generate an error because the `||` operator cannot be applied to text values.

Regular Expressions: Regular expressions can be assigned to named resources or used for *matching* (see section 3.9). Regular expressions are enclosed in `/` characters and the same escape characters are supported as for strings. References to both strings and other regular expressions may be interpolated – if a string is interpolated, then meta characters in the string will be escaped. The full Perl regular expression syntax is supported, except for variable interpolation (`$` and `@`). For example:

```
RE1 => /ab*/
String1 => 'ab*'
RE2 => /x,${RE1},${String1}/
```

compiles to:

```
RE1 => /ab*/
String1 => 'ab*'
RE2 => /x,ab*,ab\*/
```

3.2 Undefined Values

A resource value may be explicitly marked as *undefined*:

```
PathName => ?
```

This indicates that the value must be defined somewhere else in the configuration (probably in a different file). All other definitions will take priority over the default value, so this will only appear in the final configuration – where it generates an error – if no non-default value is supplied elsewhere.

3.3 Lists

A *list* may contain elements of any type:

```
Users => [john,jane]
Mixed => [3,[4,x],'foo bar']
```

3.4 Blocks

A *block* is an unordered collection of resources:

```
MailService => {
  Port => 25
  Packages => [ sendmail ]
}
```

The resource names must be unique within the block.

Sometimes it is useful to specify a single resource value in a nested block hierarchy[2]:

```
Users => {
  Students => {
    John => {
      UID => 123
    }
  }
}
```

The nested blocks here are rather clumsy and the following shorthand (an *implicit block*) can be used:

```
Users.Students.John.UID => 123
```

This is simply a syntactic shorthand for the previous example.

[2] This is particularly common when the block is to be merged with some other block (see 4).

3.5 References

A *reference* (`$`) can be used in resource values to refer to other (top-level) resources:

```
Domain => 'foo.com'
WebDomain => $Domain
```

3.6 Selectors

A *selector* (`.`) can be used to reference individual resources from a block. The selector can be applied directly to the block, but it is usually only useful when applied to a reference:

```
Ports => {
  http => 80
  ssl => 443
}
SSLPort => $Ports.ssl
```

Selectors can also be applied to lists:

```
Ports => [ 80, 443 ]
FirstPort => $Ports.0
```

Note that brackets must be used if multiple numeric selectors are specified, to prevent the selectors being interpreted as a decimal number:

```
Lists => [ [1,2], [3,4,5] ]
Two => $Lists.(0).1
```

3.7 Expressions

The operators shown in figure 1 can be used to combine resource values. The groups are listed in priority order and sub-expressions can be grouped using brackets. The second and third columns in the table show the supported argument and result types.

For example:

```
Port => 25
Memory => 6
CPU => slow
TCPPort => $Port ++ '/tcp'
BigMachine => $Memory>4 && $CPU==fast
```

3.8 Conditionals

Conditionals are supported. These are normally used in a resource value:

```
BigMachine => true
StandardSize => 50
Size => if ($BigMachine)
  then $StandardSize+100
  else $StandardSize
```

X `||` Y	bool	bool	Logical 'or'
X `&&` Y	bool	bool	Logical 'and'
$X < Y$	scalar	bool	Numerical or lexical comparison
$X <= Y$	scalar	bool	Numerical or lexical comparison
$X == Y$	scalar	bool	Numerical or lexical comparison
$X != Y$	scalar	bool	Numerical or lexical comparison
$X >= Y$	scalar	bool	Numerical or lexical comparison
$X > Y$	scalar	bool	Numerical or lexical comparison
$X =\sim Y$	text,regex	bool	Regular expression match
$X !\sim Y$	text,regex	bool	Regular expression no match
X `++` Y	scalar	scalar	Concatenation
$X + Y$	number	number	Addition
$X - Y$	number	number	Subtraction
$X * Y$	number	number	Multiplication
X / Y	number	number	Division
$- X$	number	number	Unary minus
$! X$	bool	bool	Boolean negation

Figure 1: CF2 operators

The condition must be enclosed in brackets and evaluate to a boolean value. The `else` clause is optional and will return an undefined value if not present. Only one of the conditional branches will be evaluated.

Conditionals may also be used *inline*. In this case, the result of the conditional evaluation must be a block and the resulting resources are merged with the block containing the conditional statement:

```
Type => server
if ($Type == server)
  then { Foo => 'something' }
  else { Bar => 'something else' }
```

Evaluates to:

```
Type => server
Foo => something
```

Inline conditionals are only necessary in the somewhat unusual case where different sets of resource names need to be defined in the two different branches – hence the rather contrived example. It is recommended that they are only used in this situation. The resources returned from the conditional statement must not conflict

with any references that appear in the condition, or in any other inline statement.

3.9 Regular Expressions

The operators =~ and !~ allow text values to be matched against regular expressions, returning a boolean result. This can be assigned to a resource or used directly in a conditional:

```
Addr => '129.215.45.12'
if ($Addr !~ /^\d+\.\d+\.\d+\.\d+$/)
  then fail( "invalid: ${Addr}" )
```

The `match` function allows capture groups to be extracted from the regular expression:

```
Date => '2/11/23'
Block =>
  if ($Date=~/^(\d+)\/(\d+)\/(\d+)$/)
  then {
    Day => match(1)
    Month => match(2)
    Year => 2000 + match(3)
  } else fail("invalid date: ${Date}")
```

which compiles to:

```
Date => '2/11/23'
Block => {
  Day => 2
  Month => 11
  Year => 2023
}
```

However:

1. The match function is only valid in a branch of a conditional statement which contains a regular expression directly in its condition.
2. If multiple regular expressions appear in the condition, then only one of them may contain capture groups (brackets).

These restrictions are a direct consequence of the fact that CF2 statements are not dependent on ordering. In particular, the following is not valid because the result would be different if the statements appeared in a different order:

```
Date => '2/11/23'
Valid => $Date=~/^(\d+)\/(\d+)\/(\d+)$/
Month => match(1)
```

3.10 Import

Resources may be imported from another file:

```
I => import(f1)
```

Where `f1` contains:

```
Y => 3
Z => 4
```

This evaluates to a block containing all of the resources in the imported file:

```
I => {
  Y => 3
  Z => 4
}
```

The argument specifies the configuration file to be included. This is normally a literal or string, but it may be any expression which evaluates to a text value. The compiler searches for the file in a number of locations:

- If `file` starts with a `/`, it is treated as an absolute pathname.
- If `file` starts with `./`, it is interpreted relative to the current directory.
- If `file` has the form *project*`:`*path*, the path is interpreted relative to the configuration directory of the specified project. The *project* defaults to the current project if omitted. The *path* defaults to the configuration directory of the specified project if omitted.
- If none of the above are true:
  - The current working directory is searched.
  - The environment variable `CF2PATH` is used as a (colon-separated) path of directories to search.
  - The directory of the importing file is then searched.
- Finally, the configuration directory for the current project is searched.

In all cases, a default extension of `.cf2` is assumed.

The import statement may also be used *inline*. For example:

```
File => 'f1'
import($File)
```

Where `f1` contains:

```
Y => 3
Z => 4
```

Which evaluates to:

```
File => f1
Y => 3
Z => 4
```

In this case, the resources from the imported file are merged with any other resources in the block containing the `import` statement. Resources defined in the importing block will override any resources with the same name in the imported file (the order of the `import` statements is not significant). Imported resources must not conflict with any references which appear in the argument or in any other inline statement – for example the file `f1` must not contain a (re)definition of the `File` resource.

Notice that files imported with an inline statement have the ability to modify any resource in the importing block. If the import statement is used at the top-level of the configuration, this includes any resource in the configuration. For this reason, non-inline import statements are usually preferable (see section 5).

3.11 Private Resources

It is sometimes useful to prevent certain resources from appearing in the output configuration. For example, a large file of common resources may have been imported only in order to extract one particular value - the `private` qualifier can then be used to hide the unwanted resources. This example outputs only the resource `v` whose value is the same as the resource `Res` from the file `bigfile`:

```
private BigBlock => import(bigfile)
v => $BigBlock.Res
```

The `Private` (`-p`) command line option can be used to override the `private` qualifier and output all of the resources (including the private ones).

3.12 External Functions

External functions can be written in Perl as separate modules and loaded dynamically when they are referenced from the configuration:

```
private V => [a,b]
T => upcase(typeof($V))
```

Figure 2 shows the functions provided with the standard distribution, and section 6 explains how to create custom external functions. Function names are case-insensitive.

Functions returning a block may also be used *inline*. In this case, the resulting resources are merged with the block containing the function call statement[3].

`Defined(`*X*`)`	Return `true` if *X* is defined
`DownCase(`*X*`)`	Return lower case version of text *X*
`Fail(`*X*`)`	Fail compilation with message *X*
`IPAddr(`*X*`)`	Return list of IP addresses for specified (IPV4) hostname *X*
`IPHost(`*X*`)`	Return hostname for specified (IPV4) address *X*
`Join(`*X*`,`*Y*`)`	Join list of text values *Y* with separator *X*
`Match(`*X*`)`	Return *X*th matching capture group from regular expression match
`Max(`*X*`)`	Return maximum element of list *X*
`Min(`*X*`)`	Return minimum element of list *X*
`Read(`*X*`)`	Return contents of file with pathname *X* as literal
`Sum(`*X*`)`	Return sum of elements of list *X*
`TopDir()`	Return path of top-level project directory
`TypeOf(`*X*`)`	Return type of *X*
`UpCase(`*X*`)`	Return upper case version of text *X*
`Warn(`*X*`)`	Display warning message `X`

Figure 2: CF2 standard functions

3.13 Macros

Macros provide the ability to define functions written in CF2:

```
private Root => '/home/cf2'
def fullPath(x) = $Root ++ '/' ++ $x
TestPath => fullPath(test)
LivePath => fullPath(live)
```

evaluates to:

```
TestPath => '/home/cf2/test'
LivePath => '/home/cf2/live'
```

Macros support an arbitrary number of arguments. Definitions must appear before any usage in the same file.

[3] None of the standard functions currently return a block.

4 Mutations

As described in the introduction (section 2), large configurations are created by importing separate files containing the resources for different aspects of the configuration. Typically, some resources will be specified in more than one file, and CF2 uses a *mutation function* to combine these different values and produce a single value for the final configuration.

Two standard functions are provided for the common cases (*assignment* and *merging*), and external functions can be used to implement less common, or custom mutation functions.

4.1 Assignment

Assignment (=>) is the primary mutation operator. If multiple values are defined for the same resource, then the value in the importing file will override any values specified in the imported files:

```
import(f1)
X => 1
Y => 2
```

Where `f1` contains:

```
Y => 3
Z => 4
```

Produces the following output:

```
Y => 2
Z => 4
X => 1
```

The value for `Y` in `f2` is ignored (and not evaluated).

If there is no value specified in the importing file, and different values are specified in unrelated imported files, then an error is generated and the conflict must be resolved explicitly (see section 4.4).

The assignment operator overrides all types of resource value, including lists and blocks. Note that *values in an overridden block are not "merged"*:

```
import(f2)
X => 1
Y => { A=>10, B=>20 }
```

Where `f2` contains:

```
Y => { A=>30, C=>40 }
Z => 5
```

Produces the following output:

```
Y => {
  A => 10
  B => 20
}
Z => 5
X => 1
```

This has two important properties: (1) resource values declared in a file always override values in any imported file, and (2) the order of the resource definitions and `import` statements is not significant.

4.2 Merge

The merge operator (~>) is used to merge the resources of the imported block with the resources of the importing block (it is only valid when applied to blocks). Compare this with the previous example:

```
import(f2)
X ~> 1
Y ~> { A=>10, B=>20 }
```

Where `f2` contains:

```
Y => { A=>30, C=>40 }
Z => 5
```

Which yields:

```
Y => {
  A => 10
  C => 40
  B => 20
}
Z => 5
X => 1
```

Notice that the merge operation is not recursive: The block element `Y.A` has the value `10` from the importing file and the value (`30`) from the imported file is ignored. To merge these instead of overriding them, the importing file should specify `A~>10` rather than `A=>10`.

The order in which the block elements are merged is the same as the assignment operator, so the resulting value is independent of the order of any import statements or resource definitions.

4.3 External Mutation Functions

In addition to the standard mutation functions, some external functions can be used as mutation operators (see figure 3). For example `max`:

```
import(f1)
Y ~(max)> 2
Z ~(max)> 5
```

Where `f2` contains:

```
Y => 3
Z => 4
```

Which yields:

```
Y => 3
Z => 5
```

`Max(`X`)`	Return maximum all all values
`Min(`X`)`	Return minimum of all values
`Sum(`X`)`	Return sum of all values

Figure 3: CF2 mutation functions

It is also possible to create custom mutation functions (see section 6), although this is slightly more complex than creating a normal external function.

If the mutation function is commutative, and all instances of the resource specify the same mutation function, then a value can be computed for the mutation even if the different values appear in unrelated imported files which cannot be prioritised:

```
import(f3)
import(f4)
```

Where `f3` contains:

```
X ~(sum)> 4
```

And `f4` contains:

```
X ~(sum)> 3
```

Produces the following output:

```
X => 7
```

4.4 Resolving Conflicts

The order in which resources appear in a configuration file is not significant. Mutations defined in a particular source file have a higher *priority* than those defined in any imported file, and CF2 will sort the mutations for each resource based on their priority. If it is not possible to determine the relative priority of two mutations, and the mutation functions are not commutative, then CF2 will return an error. This usually indicates that there is some kind of ambiguity in the specification which needs to be manually clarified. For example:

```
Services => {
  import(database)
  import(webserver)
}
```

Where `database` contains:

```
OsVersion => 23
MoreDBResources => ...
```

And `webserver` contains:

```
OsVersion => 24
MoreWebResources => ...
```

This fails with the error "cannot determine mutation order".

This suggests that the database and web service configurations require different versions of the OS. The resolution of this conflict depends on whether there is a common value which is acceptable, and how it should be determined. Some possibilities include:

Specifying an Explicit Version: The importing file can explicitly override the imported values:

```
Services => {
  OsVersion => 27
  import(database)
  import(webserver)
}
```

Which produces:

```
Services => {
  OsVersion => 27
  MoreDBResources => ...
  MoreWebResources => ...
}
```

This may not be ideal because if the `webserver` or `database` file is updated (to say, version 30), then the overall configuration value may no longer be sufficient.

Selecting the Highest Version: The imported files can be changed to specify a minimum value for the OS version:

```
Services => {
  import(database2)
  import(webserver2)
}
```

Where `database2` contains:

```
OsVersion ~(max)> 23
MoreDBResources => ...
```

And `webserver2` contains:

```
OsVersion ~(max)> 24
MoreWebResources => ...
```

Which produces:

```
Services => {
  OsVersion => 24
  MoreDBResources => ...
  MoreWebResources => ...
}
```

Prioritising a Specific Service: If the imported files expose their individual versions, the importing file can explicitly select the version corresponding to a particular service:

```
Services => {
  OsVersion => $Services.WebOsVersion
  import(database3)
  import(webserver3)
}
```

Where `database2` contains:

```
private DBOsVersion => 23
OsVersion => $DBOsVersion
MoreDBResources => ...
```

And `webserver2` contains:

```
private WebOsVersion => 24
OsVersion => $WebOsVersion
MoreWebResources => ...
```

Which produces:

```
Services => {
  OsVersion => 24
  MoreDBResources => ...
  MoreWebResources => ...
}
```

This is preferable to assigning a specific version in the importing file because it will track any changes to the imported (`database`) file.

Performing an Arbitrary Computation: Using the two service files from the previous example, the importing file can use arbitrary logic to determine the overall value:

```
private VW => $Services.WebOsVersion
private VD => $Services.DBOsVersion
if ($VW != $VD) then {
  warn('Mismatch '++$VW++' <> '++$VD)
  private V => max([$VW,$VD])
} else {
  private V => $vw
}

Services => {
  OsVersion => $V
  import(database3)
  import(webserver3)
}
```

Which generates the specified warning and produces the following output:

```
Services => {
  OsVersion => 24
  MoreDBResources => ...
  MoreWebResources => ...
}
```

5 Security

CF2 provides a number of features to improve the security of configurations with multiple contributors:

- Responsibility for a set of configuration parameters can be delegated to a particular user (by giving them access to a specific file) with complete control over which parameters from that file are used as part of the final configuration.
- Sensitive resources such as passwords can be incorporated at compile-time from a secure database so that they do not appear in the configuration source which may be widely available, for example in a version control system.
- Digital signatures can be required and verified on imported files.

5.1 Delegation

A typical configuration is created by importing many files containing the resources for various different aspects of the system. These are usually authored by different people. In some languages (such as LCFG), an imported file can modify any resource in the configuration. In CF2, imported files can only affect resources in the imported block. This provides a mechanism to delegate specific aspects of the configuration with the authority to modify only selected resources – *provided that the imports are appropriately structured*.

For example: imagine that we would like to delegate responsibility for setting the background colour of the login screen – `Login.Colour`. We *could* enable this by delegating control of a file (`delegated`) and including this in the main configuration:

```
import(delegated)
```

Where `delegated` contains (for example):

```
Login.Colour => green
```

However, the main configuration will contain other, critical resources – for example:

```
RootUsers ~> {jane=>...,john=>...}
import(delegated2)
```

And there is nothing to stop the owner of the delegated file modifying resources anywhere in the configuration:

```
Login.Colour => green
RootUsers ~> { hacker => ... }
```

Which results in:

```
RootUsers => {
  hacker => ...
  jane => ...
  john => ...
}
Login => { Colour => green }
```

So, it is usually preferable to import the contents of the delegated file into an isolated resource and to reference the required values from elsewhere in the configuration:

```
RootUsers ~> {jane=>...,john=>...}
private Delegated => import(delegated2)
Login.Colour => $Delegated.Login.Colour
```

Using the same delegated file as previously:

```
Login.Colour => green
RootUsers ~> { hacker => ... }
```

Resulting in:

```
RootUsers => {
  jane => ...
  john => ...
}
Login => { Colour => green }
```

Which ignores all of the imported resources except those specifically referenced.

5.2 Sensitive Resources

Some resource values in a configuration often include sensitive data such as passwords. Whilst this data needs to be present in the compiled configuration, it is usually undesirable to have these values exposed in plaintext in the configuration source. CF2 has the ability to create a block of resources automatically by importing key/value pairs from an encrypted database. Other resources can then reference values from this block so that the source itself does not contain any sensitive information.

The encrypted database must be stored in KDBX format[4]. CF2 provides no facilities itself for creating and managing the database because there are several good clients available (e.g. KeePassXC[5]) as well as API libraries (e.g. File::KDBX[6]).

Resources can be imported from a KDBX file simply by specifying the path with an appropriate extension in an import statement (if the extension is omitted, the include will search for a CF2 or a KDBX file). For example:

```
private MyPwds => import('mydb.kdbx')
FredsPwd => $MyPwds.fred.Password
```

This example requires only a single value to be extracted from the KeePass file so the imported block is marked as `private` to prevent the other values appearing in the output configuration.

The password for the database itself must be supplied manually at compile time, or stored in the OSX keychain. If an OSX keychain entry is used, this should have the service name `cf2.keepassx` and the user (account) name should be the same as the database name (with no path or extension). If there is no OSX keychain entry, then the `-P` option should be specified and the compiler will prompt for the password.

Each KeePass entry maps directly into a CF2 resource block with the following keys: `Title`, `Password`, `UserName`, `URL` and `Notes`. If the KeePass entry contains additional custom fields, these will also be included in the block. Note that the the title of the KeePass entry (and any custom field names) need to be valid CF2 identifiers - if this is not the case, then the illegal characters will be replaced with the Unicode dingbat ★. For example:

[4] https://keepass.info/help/kb/kdbx_4.html
[5] https://keepassxc.org
[6] https://metacpan.org/pod/File::KDBX

```
entry★1 => {
  Notes => ''
  Password => 'secret'
  Title => entry-1
  URL => ''
  UserName => mary
  MyCustom => 'my value'
}
```

The import will fail if any block has duplicate resource names.

KeePass groups are mapped into corresponding (nested) blocks (however, the KeeShare function of KeePassXC exports a shared database in an older KDBX format which contains a flattened copy of any subgroups in the main database).

5.3 Signed Files

The CF2 compiler can check PGP signatures on imported files against a list of permitted user identifiers (or a single user):

```
import(f1,['user1@org','user2@org'])
import(f2,'user@org')
```

The second argument will usually be a text value (or a list of text values) but it may be any expression (e.g. a reference) which evaluates to one of these.

The `ptools cf2` command has several options to configure the underlying gpg:

--gpghome - the gpg directory (~/.gnupg).

--gpguser - the gpg user.

--gpgpass - the gpg pass phrase.

Source files can be signed with a command such as:

```
gpg --sign --clearsign file.cf2
```

The resulting file will have a an additional `.asc` extension which can usually be removed.

The CF2 compiler can also sign the output:

```
ptools cf2 compile -S file.cf2
```

6 Writing Custom Functions

CF2 functions are implemented as independent Perl modules. Custom functions can be created by writing a module with the appropriate interface and placing it in a directory contained in the *function path*. The function path can be set to a colon-separated list of directories with the `FunctionPath` (`-F`) command line option. The name of the module is used as the name of the function. The modules for the standard functions are in the directory `CF2::Functions` of the distribution and these can be inspected as code examples.

All external function modules should provide a basic interface which allows them to be called as CF2 functions. They may also provide an optional additional interface which allows them to be used as mutation functions.

6.1 The Basic Interface

The basic interface includes a prototype specification which defines the types of the arguments, and an `Eval` function which performs the evaluation. For example, the following module defines a simple addition function:

```
Proto(NUMBER,NUMBER);
sub Eval($$) {
  my ($args,$ctx) = @_;
  return OK($args->[0]+$args->[1]);
}
```

The Prototype: The `Proto` function defines the number and permitted types of the arguments – two numbers in the above example. The types will be checked before the function is called, and the following types are supported: `ANYTYPE`, `UNDEF`, `BLOCK`, `LIST`, `SCALAR`, `BOOL`, `NUMBER`, `LITERAL`, `REGEX`, and `STRING`. Multiple permissible types can be specified by combining them with `|` – for example: `BOOL|NUMBER`.

The Evaluation Function: The `Eval` function is passed two arguments: the first is the list of argument values, the second is a internal *context* object – this contains information about the context of the function call, including the symbol table used to evaluate references.

By default, CF2 will evaluate the arguments and convert them into pure Perl data types (scalar, hashes or lists) before passing them to the function. Flags can be added to the types in the prototype to indicate that an argument should be passed instead as an evaluated or unevaluated AST node. For example:

```
Proto(ANYTYPE|NOEVAL,BOOL);
```

The AST nodes are internal data structures which contain additional information, including for example, the location of the term in the source file.

The result of the `Eval` function should be returned using `OK(`*result*`)` for a successful return, or `Fail(`*message*`)` to return an error. The result value may be a Perl scalar, list or hash, or an AST node. To return a boolean value, return one of the Perl functions

`True` or `False`.

6.2 Mutation Functions

In addition to the basic interface, functions which are intended to be used as mutations must provide a `Mutate` function, and an indication if the function is commutative. For example:

```
Commutative;
sub Mutate($$) {
  my ($em,$ms,$ctx,$name) = @_;
...
}
```

The interface to the `Mutate` function is more complex and is not fully described here. The arguments include the (evaluated) highest priority mutation objects (`$em`) and a list of any other (unevaluated) lower priority mutations objects (`$ms`). The function must perform any necessary evaluations and create an evaluated AST node which should be returned with the `OK()` function.

Appendix A Configuration Grammar

Configuration	:=	Block *EOF*
Block	:=	(Statement (Sep Statement) *) ?
Statement	:=	Conditional | Import | FunCall | MacroDef | ResourceDef
MacroDef	:=	`def` *IDENTIFIER* `(` (*IDENTIFIER* (Sep *IDENTIFIER*) *) ? `)` `=` Expr
ResourceDef	:=	Qualifier * *IDENTIFIER* (`.` *IDENTIFIER*) * Mutation +
Qualifier	:=	`private`
Mutation	:=	MutateOp Expr
MutateOp	:=	`=>` | `~>` | (`~(` *IDENTIFIER* `)>`)
Expr	:=	Conjunction (`||` Conjunction) *
Conjunction	:=	Comparison (`&&` Comparison) *
Comparison	:=	Match ((`<=` | `>=` | `==` | `!=` | `<` | `>`) Match) ?
Match	:=	Concat ((`=~` | `!~`) Value) ?
Concat	:=	Sum (`++` Sum) *
Sum	:=	Product ((`+` | `-`) Product) *
Product	:=	Neg ((`*` | `/`) Neg) *
Neg	:=	((`-`) | `!`) * Selection
Selection	:=	Value (`.` Value) *
Term	:=	`(` Expr `)`
Value	:=	Import | Conditional | FunCall | Undef | Scalar | RegEx | SubBlock | List | Ref | Term
FunCall	:=	*IDENTIFIER* `(` (Expr (Sep Expr) *) ? `)`
Import	:=	`import` `(` Expr (Sep Expr) ? `)`
Conditional	:=	`if` Term `then` Expr (`else` Expr) ?
Undef	:=	`?`
SubBlock	:=	`{` Block `}`
List	:=	`[` (Expr (Sep Expr) *) ? `]`
Ref	:=	`$` *IDENTIFIER*
Sep	:=	*EOL* | `,`
Scalar	:=	Bool | Number | SingleString | String | Literal
Bool	:=	`/(true)|(false)/`
Number	:=	`/\-?\d+(\.\d+)?/`
SingleString	:=	`'` ... `'`
String	:=	`"` ... `"`
RegEx	:=	`/` ... `/`
Interpolated	:=	`${` *IDENTIFIER* (`.` Value) * `}`
Literal	:=	`/[a-zA-Z\x{80}-\x{FFFFF}][a-zA-Z0-9_\x{80}-\x{FFFFF}]*/`

References

[1] P. Anderson. Towards a high-level machine configuration system. In *Proceedings of the 8th Large Installations Systems Administration (LISA) Conference*, pages 19–26, Berkeley, CA, September 1994. Usenix. https://www.nine3.org/dcspaul/pdf/LISA8_Paper.pdf.

[2] P. Anderson. *LCFG: a Practical Tool for System Configuration*, volume 17 of *Short Topics in System Administration*. Usenix Association, 2008. https://www.nine3.org/dcspaul/pdf/17_lcfg.pdf.

[3] P. Anderson. Composition in the l3 configuration language. April 2017. https://www.nine3.org/dcspaul/pdf/l3composition.pdf.

[4] P. Anderson. PFC: Functional parser combinators in Perl. October 2023. https://www.nine3.org/downloads/ptools/2.6/pfc.pdf.

[5] P. Anderson. PTools: Perl tools & applications. October 2023. https://www.nine3.org/downloads/ptools/2.6/ptools.pdf.

[6] W. Fu, R. Perera, P. Anderson, and J. Cheney. muPuppet: A Declarative Subset of the Puppet Configuration Language. In P. Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:27, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. https://www.nine3.org/dcspaul/pdf/arxiv-puppet.pdf.

[7] J. A. Hewson, P. Anderson, and A. D. Gordon. A declarative approach to automated configuration. In *Proceedings of the 2012 LISA Conference*. Usenix Association, 2012. https://www.nine3.org/dcspaul/pdf/lisa12.pdf.

[8] A. Mikoliunaite. Usability of system configuration languages: Confusion caused by ordering. Master's thesis, School of Informatics, Edinburgh University, 2016. https://www.nine3.org/dcspaul/pdf/msc_20162088.pdf.