Specification of the Fermilab Hierarchical Configuration Language

Ryan Putz

draft 3

Contents

1 Introduction

1.1 Purpose

This document provides the formal specification for the *Fermilab Hierarchical Configuration Language*, FHiCL. This specification includes several aspects of FHiCL:

- FHiCL Syntax
- FHiCL Semantics
- Canonical Value Representations

FHiCL is a customized language created for the storage of scientific parameter sets in a medium that can be easily understood and processed.

1.2 Rationale

FHiCL was developed in order to produce a standard configuration language for the storage, communication, and manipulation of scientific parameter sets.
The existence of a standard configuration language would allow for the creation of programming language bindings that can read and process valid FHiCL documents, returning a parameter set to the user.

1.3 Scope of This Facility

This project will include the development of a grammar specification for FHiCL (I.E. this document), creation of a basline parser (using Yacc and Bison), and the creation of various programming language bindings which shall read in FHiCL documents and attempt to create a parameter set.

2 FHiCL Syntax

The FHiCL syntax is defined by the following bison grammar:

```
\include{"bnf.y"}
```

In this grammar, all uppercase names denote tokens. These tokens are defined by the following flex specification:

```
\include{"bnf.l"}
```

2.1 Low-Level Entities

Note: For all rules in this section, whitespace is not allowed between tokens.

2.1.1 Reserved and Special Characters

A *char* is one of:

1. any ASCII character except for:
   - double-quote (”)
   - reverse solidus (\)
   - control characters
2. (*printable* characters)
3. one of a number escape sequences, noted below:
   - escaped double-quote (\”)
   - reverse solidus (\\)
   - solidus (\/)

   There are a number of reserved char values:

   - colon (:)
   - double colon (::)
   - left/right brace ({})
   - left/right bracket ([])
   - left/right paren (())
   - at sign (@)

2.1.2 Atoms

The most basic unit of FHiCL is the *atom*, which is defined as:

```
atom: number | string | NIL | BOOL_TOK | REF
```

EBNF:

```
atom   =>    char | string
string =>    alpha[alnum]* | digit[alnum]*
```

Notes:

- The canonical representation of an atom is a sequence of printable characters.
- Every atom can be requested in canonical string form.
- There are three valid syntaxes for a string in FHiCL:
  1. Alpha Start String - No quotes, string values must be *simple* and contain no white space.
  2. Single-Quote - Surrounded by single quotes; all content is quoted verbatim.
  3. Double-Quote - Surrounded by double quotes; content may contain special escaped characters.
- The two special characters that are allowed in *all* string forms are newline and tab.

2.2 Mid-level Entities

Note: For all rules in this section, whitespace is allowed only where specified by the whitespace token *ws*.

2.2.1 Comments

FHiCL comments are denoted by the # symbol, or by \\ which is placed at the beginning of the comment. FHiCL comments are single-line, and should be ignored by parsers.

2.2.2 Names

A *name* is similar to a key in a key-value pair of a C++ mapping, or ... (other language examples here)
Example:

```
x: 1.0
```

In this case, ”x” is a *name*.

2.2.3 Hierarchical Names

A hierarchical name, or *hname* is a compound name using the *dot index* or *bracket index* to denote levels of scope.

```
cont1:{x: 1.0 y: 2.0 z: 3.0}
cont1.x : 5
OR
cont2:[1, 2, 3}
cont2[0] : 1
```

EBNF:

```
hname => atom (DOT_INDEX|BRACKET_INDEX) atom
```

2.2.4 Values

An element of type *value* is either a single atom, a collection of atoms, or a collection of associations. Example:

```
a : 1.0
#Where "1.0" is the value of the atom named "a"
```

EBNF:

```
value => table|sequence|atom
```

Note: see definitions for *table* and *sequence* in the next section

2.3 High-Level Entities

Note: For all the rules in this section, whitespace is allowed between any two tokens, and is not significant.

2.3.1 Definitions

An element of type *definition* is used to associate a value to a name. The syntax of a *definition* is:

```
a : 1.0
```

EBNF:

```
definition => (name|hname) COLON value
```

2.3.2 Tables

Elements of type *table* are space- or line-separated collections of definitions and are denoted by (possibly empty) braces:

```
tab1:{a: 1.0 b: 2.0 c: 3.0}
```

EBNF:

```
table => LBRACE table_body RBRACE
table_body => | table_items
table_items => table_item | [table_item + "," + table_items]
table_item => definition
```

Notes:

- Tables may contain comments **IF AND ONLY IF** the table elements are line-separated.
- Comments cannot exist inbetween space-separated table elements.
- two tables are the same when their hash code is the same (the byte sequences fed into the hash must be identical).

2.3.3 Sequences

Elements of type *sequence* are comma-separated collections of values and are denoted by (possibly empty) brackets:

```
seq1:[a, b, c, d]
```

EBNF:

```
sequence => LBRACKET sequence_body RBRACKET
sequence_body => | sequence_items
sequence_items => sequence_item | [sequence_item + "," + sequence_items]
sequence_item => value
```

NOTE: Sequences **CANNOT** contain comments.

2.3.4 Documents

The *document* is the highest-level construct in FHiCL. Any implementation of a FHiCL parser processes a *document* as if it were a single string.
A *document* consists of exactly one, possibly empty, *table* such as:

```
#Document start
main:{
      a: 1.0
      b: "hi"
      c: dog
     }
#Document end
```

EBNF:

```
document => table
```

Documents may have one or more prologs at the top of the document. The only items that may occur before a prolog are comments and other prologs.

2.3.5 Overrides

An element of type *override* is used to associate an existing element with a new value, or to create a new element in a *table* or *sequence*. The syntax for an override:

```
a: 1.0 #Declaration and initialization
a : 5.0 #Override (Assignment)

OR

tab1:{ a:1 b:2 c:3 }
tab1.d : 5 #Creating a new element ’d’ in table ’tab1’

OR

seq1:[ 1, 2, 3 ]
seq1[3] : 5 #Creating a new element ’5’ in sequence ’seq1’
```

EBNF:

```
override => (name|hname|DOTINDEX|BRACKETINDEX) COLON value
```

Note: the *name* for an override is an *hname*.

2.3.6 Includes

In order to import values from external documents into a FHiCL document, an *include* statement is used to tell which file's values should be inserted into the document. A FHiCL #include statement differs from the C++ #include statement in that the FHiCL #include acts more as a union of two documents , as opposed to just allowing one file to access another.
The *include* statement syntax is as follows:

```
//This is a valid include statement:
#include "filename.ext"

//These are invalid include statements:
#include filename.ext
//include "filename.ext"
#include"filename.ext"
include "filename.ext"
#includefilename.ext
```

Where the quoted string ”filename.ext” represents the file name and file extension of the included file.
NOTES:

- There is exactly one space between '#include' and 'filename.ext'.
- Also, the filname must be enclosed in double quotes.
- Any deviation from the include statement syntax will result in a parse failure.
- Circular or repetive includes are *not* supported and should be checked for by the parser.
- Included values can be overridden and can override values that are within the same scope and share the same name.
- Includes must be on their own line, otherwise they will be treated as comments

2.3.7 Prologs

Prologs are constructs which exist at the start of a FHiCL document. Prolog boundaries are denoted by the use of *BEGIN_PROLOG* and *END_PROLOG*. All data within a prolog may not be reassociated/reassigned outside of the prolog. Data within a prolog may be referenced in the main document.
Below is an example of a valid FHiCL Prolog:

```
BEGIN_PROLOG
x :5
y :6
END_PROLOG
```

2.3.7.1 References

In order to associate a name with the value of a pre-existing definition the use of the FHiCL *reference* notation is required:

```
@local::
OR
@db::
```

Example:

```
x : 5
y : @local::x
z : @db::x
```

References point to the most-recently encountered variable with a matching name. Reference names must be extremely specific in which value they are pointing to.
For instance, if we have a table *tab1* such as:

```
tab1:{ a:1 b:2 c:3 }
```

and we want to set an outside variable to the value of *a* in *tab1*. The reference for this would look like:

```
tab1:{ a:1 b:2 c:3 }
x : @local::tab1.a
```

And this would give us a resulting parameter set of *x : 1*
In situations where an element in a prolog shares a name with an element in the document body, any references made to a variable of the same name will result in a reference look-up to the element in the document body.

3 FHiCL Semantics

3.1 High-level Result of a Successful Parse

The result of parsing a *document* is a single *table*. The *definition*s and *override*s appearing before the top-level *table* are intended to allow the user to supply values to be substituted into elements in the *table*. The *definition*s and *override*s appearing after the top-level *table* are intended to allow the user to replace values in that table.

3.2 Representation of Atoms

In the parse results, all *atom*s except for `nil` and *ref* are represented as character strings. The atom `nil` is represented by a value specified by the binding for a given programming language. The resolution of *ref*s is described in section *refs* below.

Each language binding provides its own mechanism for turning atoms of type *integer*, *real* and *complex* from their string representation into the appropriate numerical representation.

3.3 Value Semantics

- Values of true and ”true” are identical.
- Values of false and ”false” are identical.
- To include leading or trailing zeros in any number, the number must be quoted.
- The small range of a real or integer value is 1,000,000.
- Real numbers with no fration will be converted to integer format if within the small range.
- A canonical real has no leading zeros in exponent or fraction, lower case e, with plus or minus.
- Canonical integers have no leading zeros.
- Null is not supported.
- Infinity and +infinity both become ”infinity”.
- −Inifinity is supported.
- A leading + is not legal, except when used with ”infinity”
- A leading 0 is not legal unless it is the sole character. ****Under Review****
- Adding a double to a parameter set programmatically will have a rule that specifies how it will be handled.
- 00.000E+000 will be ”0.0” in canonical form.
- Any exponent as e+0 will be stripped in canonical form.
- −0.0 retains the negative.
- nil and ”nil” are treated as identical.
- String concatentation opreatiosn are permitted, but only quoted string values.
- No unquoted white space is permitted
- Quotes for string values can be left out if the string value has no white space and is simple.

3.4 Resolution of *Referencess*

Atoms of type *reference* are replaced by the value indicated by the *hname* part of the *reference*, where the environment in which the *hname* is evaluated is determined by the `db` or `local` at the end of the *reference*.

The presence of `local` indicates that the scope in which the *hname* is to be evaluated is the previously-read *document* text. The presence of `db` indicates that the scope in which the *hname* is evaluated is the single database to which the parser has access.

If the parser has no access to a database, and a *reference* which ends in `db` is encountered, a parse failure results. If, in the appropriate scope, the *hname* in a *reference* does not resolve to any *value*, a parse failure results.

3.5 Issues with Leading Zeros and Canonical Representation

As a rule, leading zeros are not allowed in any situation where a number may be misinterpreted as a non-base-10 number with the inclusion of (a) leading zero(s).

This rule only applies to numbers that may be represented as a base-10 integer. Floating point, binary, hexidecimal, and octal numbers may have leading zeros. Exponential numbers may have leading zeros, but if they are representable as a base-10 integer, their canonical form will be in integer form.

The rationale for this rule is that in some programming languages, a leading zero is used to denote a non-base-10 number, I.E. ”0x” is used to denote a hexidecimal number.

4 Features of Programming Language Bindings

4.1 Processing

Each programming language binding for FHiCL must be able to produce a parameter set in the standard FHiCL syntax.

4.2 Output

Each language binding shall return a native container construct closest to that of the FHiCL table. The returned container shall contain a valid FHiCL parameter set.

4.3 Storage

Storage of parsed results from each program language binding shall be in th standard FHiCL syntax as defined above. FHiCL documents are to be stored in files with the suffix ".fcl".

5 General Requirements

5.1 Additional Requirements for Dynamically Typed Languages

Tables and sequences should be represented by a built-in type of the programming language.
If the target programming language has a standard JSON library, we want to make sure that our constructs can be translated to JSON format and back without use of any FHiCL-specific library.
It is important that code that uses the representation of a *table* not need any FHiCL-specific code.

6 Output Requirements

6.1 Output Intended for Human Reading

"Pretty Printers" must make use of newlines and indentation throughout parameter set output. The use of newlines and indentation between table elements, individual associations, include statements and comments is required.

6.2 Output Intended for Machine Reading

Output for use by machine(s) is to be machine parsable, have an ASCII dump facility and platform neutral. Machine output is to be exclude unnecessary elements such as comments.

7 Glossary

7.1 Alphas

An *alpha* is any of the ASCII characters a-z or A-Z.

7.2 Digits

A *digit* is any of the ASCII characters 0-9.

7.3 White Space

A *ws* is one of the three whitespace characters: space/tab, newline, and line return.

7.4 Alphanumerics

An *alnum* is any of the ASCII characters a-z, A-Z, 0-9 or other *printable* characters