

# DRAFT—Specification of the Fermilab Hierarchical Configuration Language—DRAFT

Marc Paterno  
Randolph J. Herber  
draft 4

---

## Contents

<b>1 Introduction</b>	<b>1</b>
<b>2 Preprocessing</b>	<b>2</b>
<b>3 Configuration language syntax</b>	<b>3</b>
<b>4 Configuration language semantics</b>	<b>12</b>
<b>A Differences between JSON and FCL</b>	<b>13</b>
<b>B Processing notes for URI inclusion</b>	<b>14</b>
<b>Index</b>	<b>15</b>

---

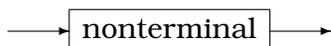
## 1 Introduction

### 1.1 Purpose of this document

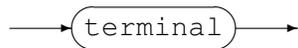
This document provides the formal specification for the *Fermilab Hierarchical Configuration Language*, FHiCL. It uses extended Backus-Naur format (EBNF) to describe the language grammar. The EBNF is presented as “railroad” diagrams. For purposes of pronunciation, the acronym rhythms with “fickle.”

### 1.2 Notation used in this document

Any *nonterminal* is formatted in a square box, like:



Any *terminal* text is formatted in a box with rounded corners, like:

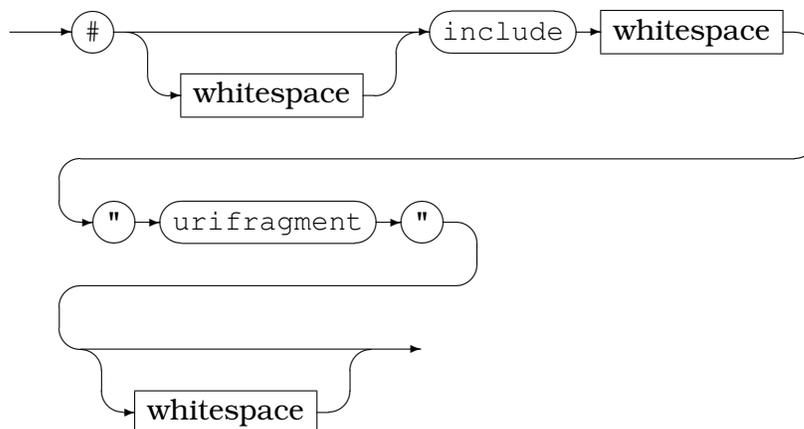


Arrows are used to show the direction in which diagrams should be read.

## 2 Preprocessing

Conceptually, preprocessing is a separate processing step which occurs before the tokenization step and which may be and probably will be implemented in a separate program.

*preprocessorlines*



Preprocessor lines are syntactically hash comments in the FHiCL. When discovered, the processor line will be copied to the output and then followed by the contents of the urifragment data entity. Inclusions may be nested to a minimum depth of 10. If there exists a FHiCL shell environment variable, which is a comma separated list of URI headers, then a URL will be formed with each URI header from left to right concatenated with a '/' (if the fragment does not start already with a '/' and the urifragment. The first successful URL will be used. If no URL is successful, then the preprocessor line produces no output and no error indications. If there does not exist a FHiCL shell environment variable, then the urifragment will be interpreted as a file name relative to the current working directory if it does not begin with a '/' and as an absolute file name if it does begin with a '/'.

## 3 Configuration language syntax

### 3.1 Encodings

The input will be encoded in ASCII, a seven bit code. In eight-bit bytes, the high-order bit will be zero. In strings, all 256 eight-bit codes may be presented using ASCII characters. A pretty printer program may represent non-printable bytes by octal or hexadecimal escape sequences. The using application is responsible for recognizing the format of such texts as binary data or wide character texts.

### 3.2 Tokenization

Conceptually, tokenization is done by selecting the token type name and string as the *first occurring longest match* in the following list of patterns. All patterns are anchored at the beginning of the remaining text. All *whitespace*, *hashcomments* and *doublesoliduscomments* are discarded as soon as they are recognized. The token type name and string are placed in a tuple. A vector of tuples is formed.

The FHiCL is whitespace delimited and is not line oriented, except for the *hascomments* and *doublesoliduscomments*.

There are seven keyword tokens: `true`, `false`, `infinity`, `PROLOG`, `END`, `nil` and `null`.

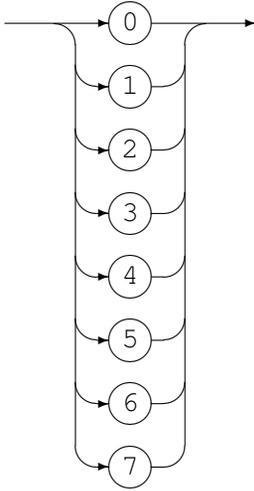
The keywords `nil` and `null` will be mapped internally to a single appropriate value for the implementation language.

N.B., leading and trailing zeroes on numeric values are considered significant and are preserved by the language.

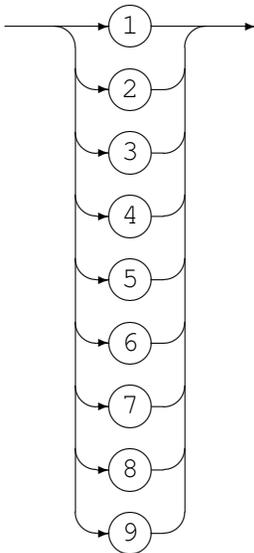
If an error token is recognized, then the input is invalid.

A list of sub patterns to simplify the main list of type names and patterns:

*octaldigit*



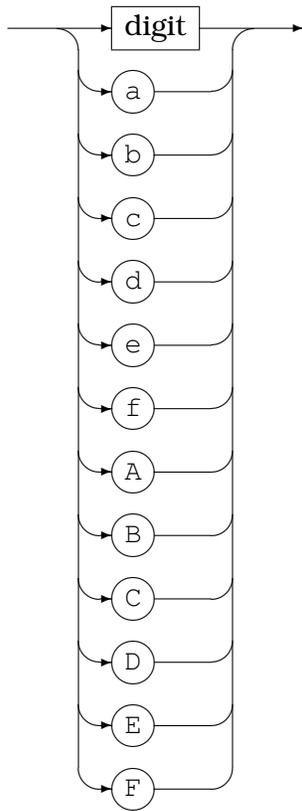
*nonzerodigit*



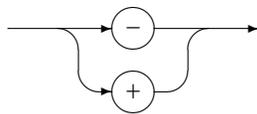
*digit*



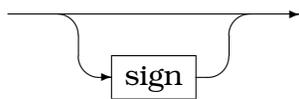
*hexdigit*



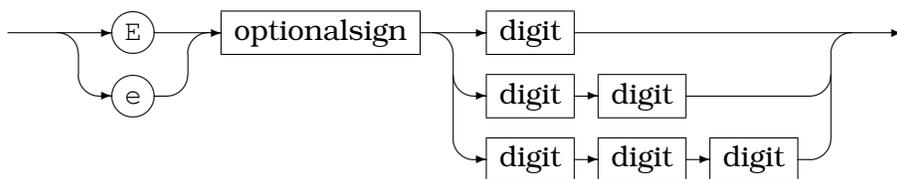
*sign*



*optionalsign*



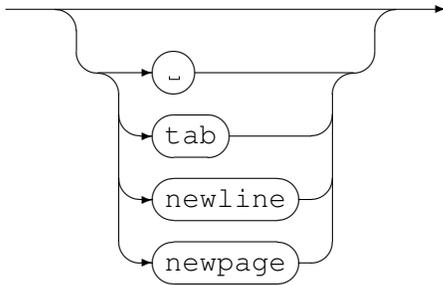
*exponent*



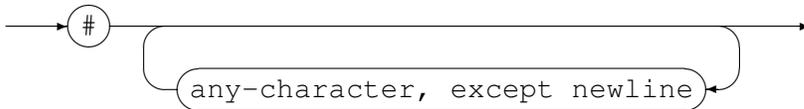
— (???) —

The list of type names and patterns:

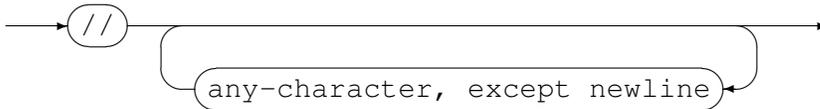
*whitespace*



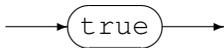
*hashcomments*



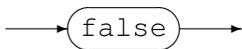
*doublesoliduscomments*



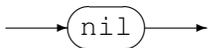
*true*



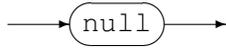
*false*



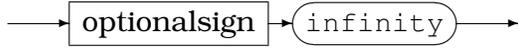
*nil*



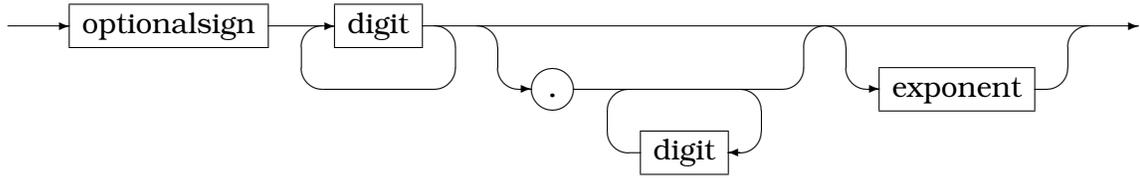
*nil*



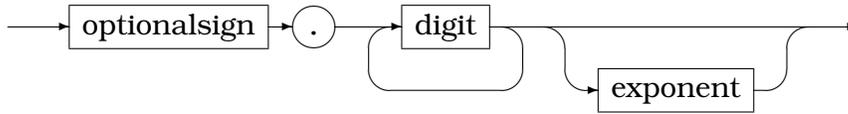
*number*



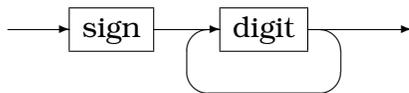
*number*



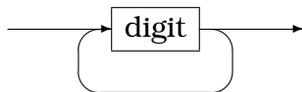
*number*



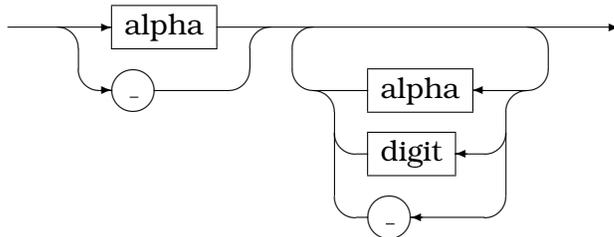
*number*

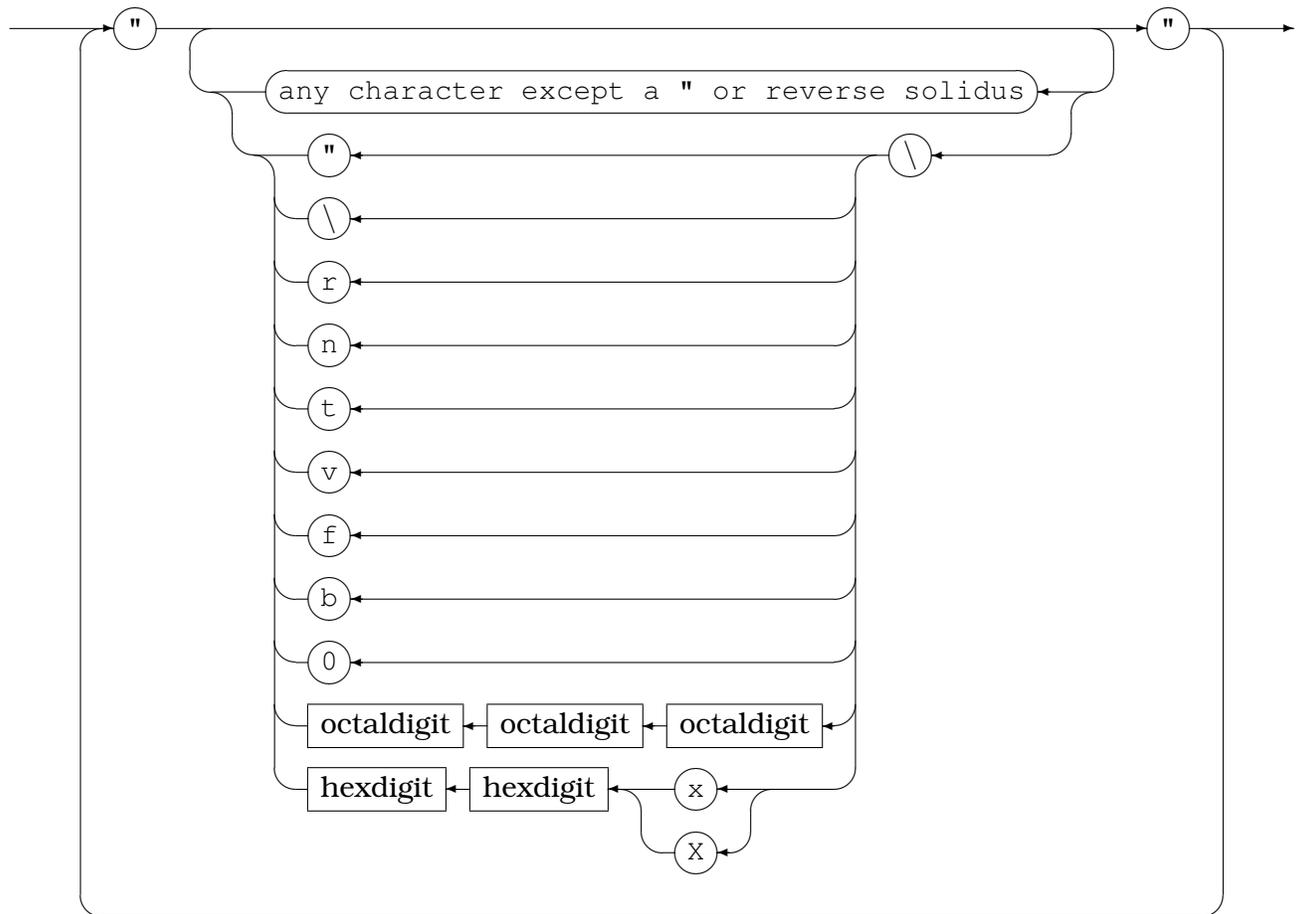
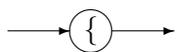


*ordinal*

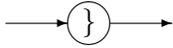


*name*



*string**dot**atsign**leftbrace*

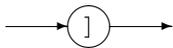
*rightbrace*



*leftbracket*



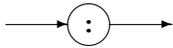
*rightbracket*



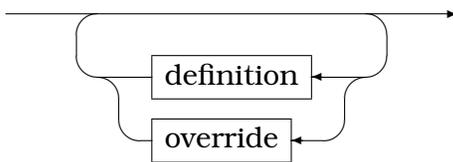
*comma*



*is*



*specifications*



*error*



### 3.3 High-level entities

Whitespace, hashcomments and doublesolidus comments are dropped when recognized; therefore, they are not significant. They will not be mentioned in the following grammar rules.

A sequence of strings will be concatenated.

PROLOG keyword and PROLOG END keyword sequences, if present, must occur in complete sets. All definitions occurring within such a set are omitted from the output.

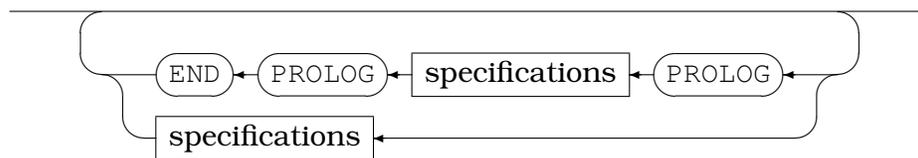
The *document* is the highest-level construct in FCL. Any implementation of an FCL parser processes a *document* as a single string.

An empty output is acceptable.

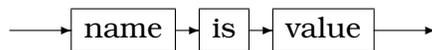
Null is an appropriate distinct value of the parser implementation language.

The output of a successful parse of a document is a table of definitions. Overrides are not included in the output table. An override which does not find its search target is an error. If multiple copies of a name occur, then a later occurrence overrides an earlier occurrence. An implementation may issue warning messages detailing such overrides. A user may choose to treat such warnings as errors.

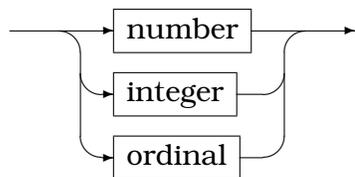
### *document*



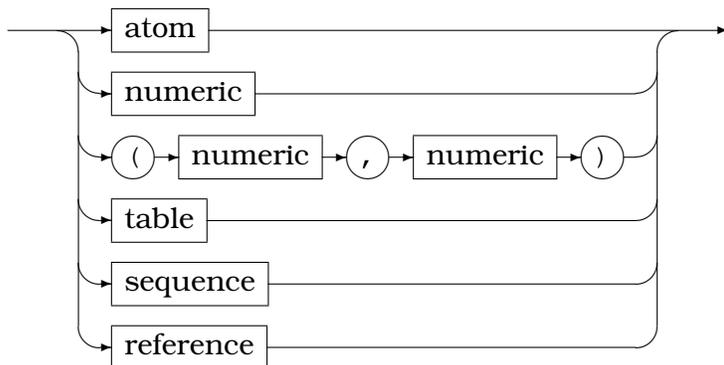
### *definition*



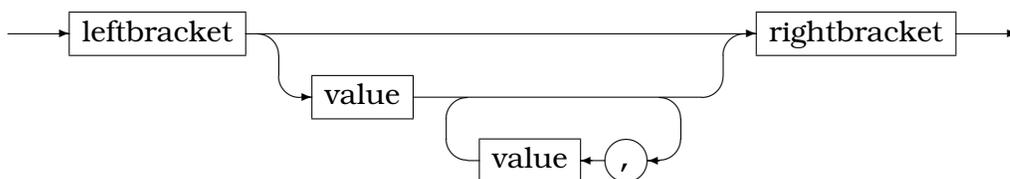
### *numeric*



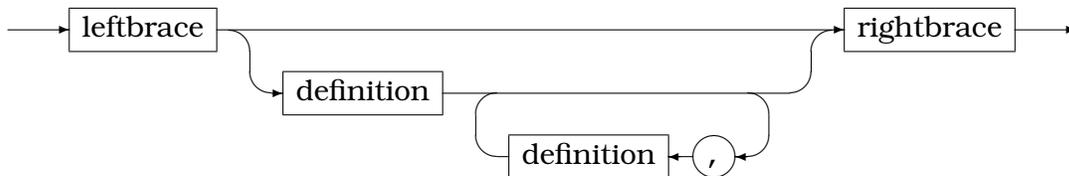
*value*



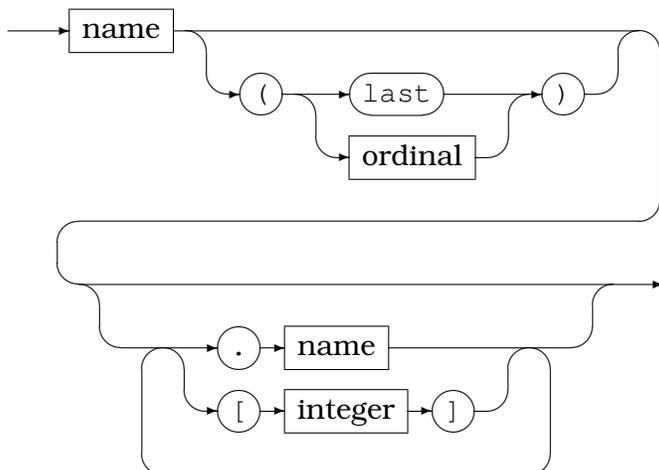
*sequence*

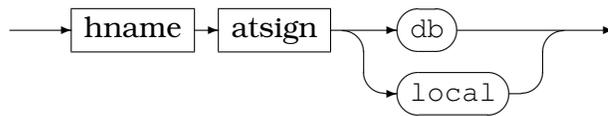
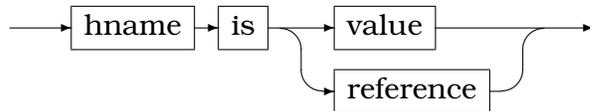


*table*



*hname*



*reference**override*

A *reference* is replaced by the value referenced. It is an error if the value referenced does not exist.

An element of type *override* is used to change the value of an existing element in the vector of tuples already built. The tuple being modified must be named. The version number indexes into a subvector of tuples with that name. If the override extends a sequence and there are missing intermediate values, then the intermediate values are set *.nil*.

In an *hname*, the optional field immediately after the initial *name* field supplies a version value. If omitted, then there must be only referencable value by that name and that is the referenced value. “(last)” means the highest available version.

## 4 Configuration language semantics

### 4.1 High-level result of a successful parse

The output of a successful parse of a document is a vector of name and value tuples. The user of FCL has a choice of which values in the vector are significant.

When the vector or a value is *pretty printed*, if an atom meets the pattern of a numeric, then it is printed without quotation, regardless of its original quotation. In the vector of tuples, all atoms are strings. Complex numbers are formed into strings as well.

The implementation must process “@local” references itself. The implementation may process “@db” references as it chooses, including considering them to be errors.

### 4.2 Representation of atoms

In the parse results, all *atoms* except for `null` and *references* are represented as character strings. The atom `null` is represented by a value specified by the binding for a given programming language. The resolution of *references* is described in section 4.3 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.

### 4.3 Resolution of *refs*

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

The presence of `here` indicates that the scope in which the *hname* is to be sought 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 *ref* which ends in `db` is encountered, a parse failure results. If, in the appropriate scope, the *hname* in a *ref* does not resolve to any *value*, a parse failure results.

## A Differences between JSON and FCL

The language described is, by intention, similar but not identical to the Javascript Object Notation (JSON), described at <http://www.json.org>.

Where JSON uses the name *object*, we use the name *table*. Where JSON uses the name *array* we use the name *sequence*.

The configuration language differs from JSON in several ways. We draw special attention to the following.

1. JSON requires that the names of members in an object be strings, which in JSON are always delimited by double quotes. In the configuration language, names of members of objects are not quoted, and are subject to different constraints; approximately stated, names must be suitable as variable names in commonly-used programming languages. See the grammar specification below for an exact description of the constraints.
2. JSON allows documents to contain any Unicode character. The configuration language restricts documents to contain only printable ASCII characters. This choice was made for the configuration language because some of the languages for which we require bindings do not have convenient support for Unicode.
3. JSON does not directly support complex numbers. The configuration language has direct support for specification of values that are complex numbers.
4. JSON recognizes *number* as a primitive data type. In the configuration language, numbers and strings are united into a common type *atom*. This choice was made for the configuration language because we need to support producing a printed representation for every value that is identical to the representation in the configuration document.

## B Processing notes for URI inclusion

1. Use recursion.
2. Arguments are:
  - a) File-like object to receive the output.
  - b) URL to be processed.
  - c) Stack of currently in-process URLs.
3. If URL to be processed already is in stack, then return (possibly with error indication).
4. Add URL to stack top.
5. Open URL, using search procedure described in *Preprocessing*.
6. If URL open failed, then return (possibly with error indication).
7. Set current position to zero.
8. Processing loop:
  - a) Copy lines, adding line length to position.
  - b) When a include line is discovered:
    - i. Close URL reading.
    - ii. Call this routine with:
      - A. File-like object to receive the output.
      - B. The just discovered URL to be processed.
      - C. Stack of currently in-process URLs.
    - iii. Handle or ignore any error
    - iv. Reopen current URL and read to current position.
9. Pop current URL from stack.
10. Return normally to caller.

## **Index**

*atsign*, 8  
*comma*, 9  
*definition*, 10  
*digit*, 4  
*document*, 10  
*dot*, 8  
*doublesoliduscomments*, 6  
*error*, 9  
*exponent*, 5  
*false*, 6  
*hashcomments*, 6  
*hexdigit*, 4  
*hname*, 11  
*is*, 9  
*leftbrace*, 8  
*leftbracket*, 9  
*name*, 7  
*nil*, 6  
*nonzerodigit*, 4  
*number*, 7  
*numeric*, 10  
*octaldigit*, 3  
*optionalsign*, 5  
*ordinal*, 7  
*override*, 12  
*preprocessorlines*, 2  
*reference*, 11  
*rightbrace*, 8  
*rightbracket*, 9  
*sequence*, 11  
*sign*, 5  
*specifications*, 9  
*string*, 7  
*table*, 11  
*true*, 6  
*value*, 10  
*whitespace*, 6