ORNL/TM-2017/698

Workbench Analysis Sequence Processor

Robert A. Lefebvre
Brandon R. Langley
Jordan P. Lefebvre

October 2017

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2017/698

Reactor and Nuclear Systems Division

WORKBENCH ANALYSIS SEQUENCE PROCESSOR

Robert A. Lefebvre
Brandon R. Langley
Jordan P. Lefebvre

Date Published: October 2017

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-BATTELLE, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

ABSTRACT

The Workbench Analysis Sequence Processor (WASP) was developed to streamline lexing, parsing, access, validation, and analysis of ascii text files.

The foundation of WASP resides on the parse tree data structure, where each node in the tree represents the syntax of the input document. Nodes can parent nodes with children. Nodes that have no children are known as *terminal* or *leaf nodes*, and they represent tokens (string, number, delimiter, etc.) in the text file. The fast lexical analyzer generator (flex - https://www.gnu.org/software/flex/) and GNU Bison parser generator (https://www.gnu.org/software/bison/) are extensively used for lexing and parsing.

1. COMPONENTS

WASP is composed of the following primary components:

1. **Core**: the waspcore package contains most necessary data structures and interface classes needed to interact with text files.
   - StringPool: a string storage optimization class where ascii data are stored in a contiguous memory block where each string is null terminated and indexed.
   - TokenPool: a token/word storage optimization class where Token information (string data via StringPool, file location) is stored. Line and column are calculated on the fly via token file offset and file line offset.
   - TreeNodePool: a TreeNode storage class where TreeNode information (token, name, parent, type, children, etc.) is stored.
   - Interpreter: an interface and high-level implementation class which facilitates specific grammar, lexer, and parser state information and parse tree storage
   - wasp_node: enumerated token/node types used to aid in identifying context and intent.
   - utils: contains utility functions useful for string processing and tree visiting.
   - wasp_bug: contains software quality assurance and development aids that can be preprocessed out of deployments.
     - design by contract: insist, require, ensure, assert, check, remember.
     - timing: 3 levels of timers for code performance monitoring. 1–3, highest to lowest.
     - debug lines: set of macros that allow printing debug information to screen.
2. **Expr**: the waspexpr package contains lexer, parser, and evaluation logic for mathematical expressions.
   - Basic mathematical operators:
     - multiplication '*'
     - division '/'
     - addition '+'
     - subtraction '-'
     - boolean ('<','<=','==','!=', etc.)
     - exponentiation '^'
   - Scalar variable assignment, reference, and creation: known variable can be referenced and updated, or new variables created during expression evaluation
   - Mathematical functions

3. **GetPot**: the waspgetpot package contains lexer, parser, and tree node view for the getpot grammar (http://getpot.sourceforge.net/).
4. **HIVE**: the Hierarchical Input Validation Engine contains algorithms for validating a parse tree using a document schema/definition file, the flexible scalar and referential rules - supporting
   - element occurrence,
   - value,
   - child uniqueness and choice,
   - existence,
   - sum,
   - predicated sum, etc.
5. **JSON**: the waspjson package contains a lexer, as well as a parser for the JSON grammar (http://www.json.org/)
6. **SIREN**: the Sequence Input Retrieval Engine (SIREN) contains a lexer, a parser, and evaluation logic for tree node lookup; the flexible tree node lookup mechanism supports
   - absolute and relative wild-carded names and value, or
   - index-predicated node path lookup
7. **SON**: the Standard Object Notation (SON) waspson package contains the lexer, parser, and tree node view for the SON grammar; the flexible, structured input entry mechanism supports
   - objects, arrays, and keyed values, as well as
   - identified objects, arrays, and keyed values.
8. **DDI**: the Definition-Driven Interpreter (DDI) contains a lexer, parser, and an interpreter for the lightweight input format.
   - Hierarchical Input format with very little syntax.
9. **HALITE** - the Hierarchical Input Template Expansion engine provides a data-driven means of expanding patterned input.
   - Supports attribute and expression evaluations.
   - Supports template imports.
   - Supports conditional action blocks.
10. **Utils** - the wasputils package contains executable utilities for listing/viewing, selecting, validating, and transforming WASP-supported grammars.
    - List: lists paths to each file element.
    - Select: allows using SIREN expression to select pieces of input.
    - Valid: validates a given text file with a given document definition/schema.
    - XML: translates a given text file into XML with data and location information

2. GETTING STARTED

For individuals wanting to compile the code from source, below are the tested requirements and configurations.

2.1 REQUIREMENTS

- C/CXX compiler (See repository .gitlab-ci.yml for build configurations)
- GCC-4.8 tested on Linux or Mac OS
- LLVM-7.0.2 tested on Mac OS
- Visual Studio 2012 for Windows
- Intel 15 for Windows
- MinGW 4.8.5 for Windows
- Git 1.7+
- CMake-2.8.12.2, 3.5, 3.8
- Python-2.7

2.2 CODE CONFIGURATION AND COMPILATION

- Save the ssh-key in code-int.ornl.gov.
- Clone wasp `git clone git@code-int.ornl.gov:lefebvre/wasp.git ~/wasp`
- Change directory into wasp `cd ~/wasp`
- Clone TriBITS `git clone https://github.com/lefebvre/TriBITS.git TriBITS` TriBITS documentation
- Clone extra repos `./TriBITS/tribits/ci_support/clone_extra_repos.py`
- Create a build directory `mkdir -p ~/build/wasp`
- Change into the build `cd ~/build/wasp`
- Create a configuration script in ~/build/. Let's call it ../configure.sh(linux)

```
#!/bin/bash
# Linux bash file example
rm -rf CMake*
cmake \
 -D CMAKE_BUILD_TYPE:STRING=RELEASE \
 -D wasp_ENABLE_ALL_PACKAGES:BOOL=ON \
 -D wasp_ENABLE_TESTS:BOOL=ON \
 -D CMAKE_INSTALL_PREFIX=`pwd`/install \
 -G "Unix Makefiles" \
 ~/wasp
```

for example, to a script that will enable getpot

```
#!/bin/bash
# Linux bash file example
rm -rf CMake*
cmake \
 -D CMAKE_BUILD_TYPE:STRING=RELEASE \
 -D wasp_ENABLE_waspgetpot=ON \
 -D wasp_ENABLE_TESTS:BOOL=ON \
 -D CMAKE_INSTALL_PREFIX=`pwd`/install \
```

```
 -G "Unix Makefiles" \
 ~/wasp
```

- Invoke configure script in the build directory. ../configure.sh or ..\configure.bat
- It is recommended that the configure script be placed in the build directory as opposed to the build/wasp directory because it allows the deletion of the build/wasp directory without removing the configuration script.

Additionally, a script to enable a third part library (TPL) plugin, configure_tpl.sh :

```
cmake \
 -D wasp_ENABLE_ALL_PACKAGES:BOOL=ON \
 -D CMAKE_BUILD_TYPE:STRING=RELWITHDEBINFO \
 -D wasp_ENABLE_INSTALL_CMAKE_CONFIG_FILES:BOOL=ON \
 -D wasp_ENABLE_googletest:BOOL=OFF \
 -D wasp_ENABLE_testframework:BOOL=OFF \
 -D wasp_ENABLE_wasppy:BOOL=OFF \
 -D wasp_ENABLE_TESTS:BOOL=OFF \
 -D CMAKE_INSTALL_PREFIX=`pwd`/install \
 -D wasp_GENERATE_EXPORT_FILE_DEPENDENCIES:BOOL=ON \
 -D CMAKE_C_COMPILER:STRING=gcc \
 -D CMAKE_CXX_COMPILER:STRING=g++ \
 -VV \
 $*
```

Subsequently, invoke the script with a path to the root source:

```
../configure_tpl.sh /path/to/source
```

Lastly, due to an issue in cmake install file creation, a manual copy of the waspConfig_install.cmake is needed:

```
cp waspConfig_install.cmake install/lib/cmake/wasp/waspConfig.cmake
```

After configuration is complete, conduct the compilation via the make system available (make, NMake, Ninja, MSBuild, etc.)

3. CORE PACKAGE

The core package of WASP provides the foundation for text processing. A primary mission of WASP is to capture information in a manner that facilitates reconstituting a user's text file. This information is captured in two primary stages: lexing and parsing of the text files:

- Lexical analysis processes the text file by recognizing patterns of text and producing a token that encapsulates information pertaining to this pattern.
- The parser recognizes patterns of tokens and constructs a parse tree for future examination.

Typically, all whitespace characters are not captured in the parse tree, as they can be deduced and reconstituted upon request.

3.1 CORE PARTS

The primary components of wasp core are:

1. StringPool: provides contiguous string storage. All strings are stored, null-terminated, in a single data container.
2. TokenPool: provides contiguous token storage. Tokens consist of string data, a token type, and a file byte offset.
   - An index is used to identify the location in the StringPool where the data reside.
   - The token pool stores new line offsets.
   - The file byte offset is used to compute line and column with the assistance of the new line offset.
3. TreeNodePool: manages TreeNodes, each of which consists of a node name, type, and child indices.
   - Uses StringPool for contiguous node name storage.
   - Uses TokenPool to store all leaf node token information.
4. Format: utility methods for formatting values which provide a type-safe printf as needed by the expression engine and HALITE.
5. wasp_node: central location for node and token type enumeration.
6. Object: generic type data structure to facilitate typed-data access to hierarchical data; facilitates HALITE data-driven capabilities
7. Interpreter: base class providing boilerplate lexing and parsing logic; contains core token and node construction logic for lexer and parser to use.

Each Pool and its subsequent Interpreter is a templated class allowing space consolidation when application size is known. For example, if the application is to interpret files that will never be more than 65KiB, an `unsigned short` can be used as the template type.

3.2 STRING POOL

The string pool consists of two members: (1) a vector of chars and (2) a vector of indices indicating string starts. This ensures that the string data consumed from a text file are reasonably maintained and that storage size is not inflated.

In a benchmark of one application's input—consisting of 300MB in which the document tokens' mathematical mode was 3 characters, with a mean of 4—using std::string produces on average ~28+ byte

overhead per token, or specifically, 8 byte heap pointer, 8 byte size, 8 byte heap page header, and 8 byte heap memory page.

In contrast, the StringPool only requires a 5 byte overhead per token, or specifically, a 4 byte index, and a null terminating character. Using the StringPool facilitates a significant memory consolidation.

3.3 TOKEN POOL

Token pool associates a token type and a file byte offset with the text that resides in the StringPool. New lines are captured as a special piece of meta data for text location deduction. The type information indicates whether it is an integer, real, word, declarator, terminator, etc. This information can be used to deduce context or perform operations on a class of data.

The file byte offset is the absolute location in the file at which the text begins. This is not intuitive to a user, but when combined with new line offset text location, line, and column, it can be deduced. Specifically, the line can be computed as the distance from the upper bound of the token's file offset in the list of line file offsets and the line offset.

```
line = distance( line_file_offsets.begin(), line_file_offsets.upperbound(
token_file_offset ) )
```

The column can be computed as the difference of the token's file offset and the upper bound, minus 1, of the token index into the list of line file offsets.

```
column = token_file_offset - (
line_file_offsets.upperbound(token_file_offset) - 1)
```

3.4 TREE NODE POOL

The tree node pool coordinates access and storage to nodes. Nodes can be classified as inner nodes and leaf nodes. Inner nodes can have children and represent a set of input. Leaf nodes always represent a token. For example, `key = 3.14159` has the following hierarchy:

```
document
|_keyed_value
  |_ declarator ('key')
  |_ assign ('=')
  |_ value  (3.14159)
```

Here, the inner node is "keyed_value," with leaf child nodes `declarator`, `assign`, and `value`.
All nodes have associated metadata:

1. Node type: declarator, terminator, value, name, etc.
2. Node name: user familiar name
3. Parent node pool index: the location in the tree node pool that the parent of the node resides

Additional metadata for parent/inner nodes consists of:

1. First child pool index, which is the index of the first child
2. The number of children

There is a convenient TreeNodeView class that provides consolidated per-node data access.

3.5 INTERPRETER

The interpreter is the base class to facilitate all syntax specific interpreters (DDI, SON, etc.). The interpreter brokers transactions between the lexer and parser, and it also stages and stores the parse tree for future access.

The interpreter manages the TreeNodePool and tracks the root of the parse tree. It also provides a stage construct to facilitate text syntax where hierarchy is ambiguous and sub-trees may not exist to immediate parent (see DDI for active use).

4. EXPRESSION ENGINE

The expression engine facilitates numerical and string expression evaluations that are integrated into other Workbench Analysis Sequence Processor components.

4.1 ARITHMETIC AND ALGEBRAIC OPERATORS

The Expr engine supports regular arithmetic and algebraic operations.

Operation	Result
Subtraction	`3 - 4` equals `-1`
Addition	`5 + 3` equals `8`
Division	`8 / 2` equals `4`
Multiplication	`3 * 3` equals `9`
Exponentiation	`2 ^ 3` equals `8`
Parenthesis	`(3 - 4) * -1` equals `1`

4.2 RELATIONAL OPERATORS

Relational operations are also supported.

Operation	Result
Equal	`3==3` equals `true`
Not Equal	`3!=3` equals `false`
Less Than	`8 < 9` equals `true`
Less Than or Equal	`8 <= 9` equals `true`
Greater Than	`9 > 8` equals `true`
Greater Than or Equal	`9 >= 8` equals `true`

Syntactic alternatives exist.

Operation	Result
Equal	`3 .eq. 3` equals `true`
Not Equal	`3 .neq. 3` equals `false`
Less Than	`8 .lt. 9` equals `true`
Less Than or Equal	`8 .lte. 9` equals `true`
Greater Than	`9 .gt. 8` equals `true`
Greater Than or Equal	`9 .gte. 8` equals `true`

4.3 BOOLEAN OPERATORS

Operation	Result
Not	`!(3==3)` equals `false`
Or	`!(1==1) || 1==1` equals `true`
And	`!(1==1) && 1==1` equals `false`

4.4 DEFAULT VARIABLES

The default variables are available for use in expression evaluations:

Name	Value
`pi`	`3.14159265359` approximately `pi`
`e`	`2.71828182845904523536028747135 27` approximately `e`
`nl`	'\n'

4.5 DEFAULT FUNCTIONS

The Expr engine also has the following functions available for use in expression evaluations:

Function	Description
`sin(theta)`	sine of theta: opposite over hypotenuse
`sinh(x)`	hyperbolic sine of x
`asin(x)`	arc sine of x
`asinh(x)`	inverse hyperbolic sine of x
`cos(theta)`	cosine of theta: adjacent over hypotenuse
`cosh(x)`	hyperbolic cosine of x
`acos(x)`	arc cosine of x
`acosh(x)`	inverse hyperbolic cosine of x
`tan(theta)`	tangent of theta: opposite over adjacent
`tanh(x)`	hyperbolic tangent of x
`atan(x)`	arc tangent of x
`atan2(x,y)`	arc tangent of x/y
`atanh(x)`	inverse hyperbolic tangent of x
`sec(theta)`	secant of theta: hypotenuse over adjacent
`csc(theta)`	cosecant of theta: hypotenuse over opposite
`cot(theta)`	cotangent of theta: adjacent over opposite

`floor(x)`	closest integer value below x
`ceil(x)`	closest integer value above x
`exp(x)`	e raised to x
`log(x)`	natural log (base e) of x
`lg(x)`	binary log (base 2) of x
`log10(x)`	common log (base 10) of x
`sqrt(x)`	square root of x
`deg2rad(x)`	converts x degrees into radians
`rad2deg(x)`	converts x radians into degrees
`deg2grad(x)`	converts x degrees into gradians
`grad2deg(x)`	converts x gradians into degrees
`round(x)`	rounds x to the closest integer
`round(x,p)`	rounds x to the p decimal point
`abs(x)`	absolute value of x
`pow(x,y)`	x raised to the power of y
`mod(x,y)`	modulo of x given y
`max(x,y)`	maximum of x or y
`min(x,y)`	minimum of x or y
`fmt(x,format)`	format the variable x with the desired format

4.6 SPECIAL FUNCTIONS

The Expr engine has a few special functions that are always available.

4.7 ARRAY ACCESS

The Expr engine supports accessing array elements by recognizing `array[index]` patterns. Only zero-based rank 1 arrays are supported.

Function	Description
`if(condition,if-true,if-false)`	if the `condition` evaluates to true, the `if-true` value is returned, else `if-false` is returned
`defined('name')`	return true, if and only if a variable with name `name` exists. Note the argument is quoted
`size(array)`	acquires the size (element count) of the given array

5. HIERARCHICAL INPUT VALIDATION ENGINE (HIVE)

The Hierarchical Input Validation Engine (**HIVE**) uses a set of rules to describe the schema of an application's input. These rules describe scalar and relational input restrictions. They can use a ***Sequence Input Retrieval Engine (SIREN) Expression***. Applications use HIVE and schema files to facilitate input validation, introspection, and input creation assistance. SIREN Expressions, SON Syntax, and Template Files are beyond the scope of this section.

The section layout is as follows:

- The Input Validation Rules Summary section provides brief descriptions of input validation rules. These rules do not contain defaults. The rules are only used when they have been specified.
- The Input Validation Details and Examples section provides a more detailed description, examples, and exact syntax of input validation rules. This section supplies an example schema, an example input that will pass validation against the schema, an example input that will fail validation against the schema, and the validation messages that HIVE produces when validating the failing input against the provided schema. If the user is incorporating a specific rule in the integration of an application, then the examples section for that particular rule should be fully understood syntactically and semantically.
- The Input Assistance Details section provides descriptions and details of the rules that may be used by input generation applications for input assistance and autocompletion.

In this document, the term ***input*** is used when referring to a file that is to be validated, and ***schema*** is used when referring to the file that describes the definition and rules against which the input is validated. Currently, schema files must be written in the SON syntax, which is used herein for example input files.

5.1 INPUT VALIDATION RULES SUMMARY

- ***MinOccurs***: describes the minimum number of times that an element is allowed to appear under its parent context.
- ***MaxOccurs***: describes the maximum number of times that an element is allowed to appear under its parent context.
- ***ValType***: describes the allowed value type for the element (Int, Real, String).
- ***ValEnums***: describes a list of allowed value choices for the element.
- ***MinValInc***: describes the minimum inclusive value that this element is allowed to have if it is a number (the provided input value must be greater than or equal to this).
- ***MaxValInc***: describes the maximum inclusive value that this element is allowed to have if it is a number (the provided input value must be less than or equal to this).
- ***MinValExc***: describes the minimum exclusive value of the element in the input if it is a number (the provided input value must be strictly greater than this).
- ***MaxValExc***: describes the maximum exclusive value of the element in the input if it is a number (the provided input value must be strictly less than this).
- ***ExistsIn***: describes a set of lookup paths into relative sections of the input file and possible constant values where the value of the element being validated must exist.
- ***NotExistsIn***: describes a set of lookup paths into relative sections of the input file where the value of the element being validated must not exist.

- ***SumOver***: describes what sum the values must add to under a given context.
- ***SumOverGroup***: describes what sum the values must add to under a given context when grouped by dividing another input element's value by a given value.
- ***IncreaseOver***: describes that the values under the element must be increasing in the order that they are read.
- ***DecreaseOver***: describes that the values under the element must be decreasing in the order that they are read.
- ***ChildAtMostOne***: describes one or more lists of lookup paths into relative sections of the input file (and possible values) where at most one is allowed to exist.
- ***ChildExactlyOne***: describes one or more lists of lookup paths into relative sections of the input file (and possible values) where at exactly one is allowed to exist.
- ***ChildAtLeastOne***: describes one or more lists of lookup paths into relative sections of the input file (and possible values) where at least one must exist.
- ***ChildCountEqual***: describes one or more lists of lookup paths into relative sections of the input file where the number of values must be equal.
- ***ChildUniqueness***: describes one or more lists of lookup paths into relative sections of the input file where the values at all of these paths must be unique.

5.2 INPUT VALIDATION DETAILS AND EXAMPLES

5.2.1 Miscellaneous Details and Examples

Before exploring the details of all of the validation rules, the first thing to note is that the hierarchy of a schema file must represent a union of all possible input hierarchies. This is just to say that every hierarchical node in an input file that is to be validated must have an exact mapping to a node at the same hierarchical path in the schema. If there is an element in an input file that does not have an exact mapping to an associated node in the schema, then that element is said to be invalid. Once the hierarchy of the schema is built, then rules can be added to every element for validation. Every element in the input document is represented by a SON object in the schema. All rules for an element at a given context are represented by either SON flag-values or SON flag-arrays.

Hierarchical nodes in the schema that do not have an associated node in the input are not traversed further. For example, if a schema defines nodes **A**, **B**, and **C** at the root level, but a given input only contains nodes **A** and **C** at its root level, then the rules directly inside of node **B** are examined to check if **B** is a required portion in the input. However, the children of node **B** are not traversed further, because it has been verified that those children are not in the input.

At the basic level, there are two types of validation messages that may be reported by HIVE.

1. Problems with the ***input*** file are reported in the form given below:

   ```
   line:X column:Y - Validation Error: MESSAGE
   ```

   Input validation applications may capture the line and column reported so that the offending input elements can be navigated to simply by clicking on the message.

2. Problems with the ***schema*** file are reported in the following form:

   ```
   Validation Error: Invalid Schema Rule: MESSAGE line:X column:Y
   ```

This message example denotes an actual error in the schema file (not the input) at the provided line and column number.

Schema example:

```
test{

    should_exist_one{
    }
    should_exist_two{
        value{
        }
    }
    invalid_rule{
        inside{
            BadRuleName=10
        }
    }

}
```

Input the example that **PASSES** validation on schema above:

```
test{

    should_exist_one    = 1
    should_exist_two    = [ 2 3 4 5 ]

}
```

Notes: This input passes validation against the provided schema because both input elements (i.e., `test/should_exist_one` and `test/should_exist_two`) exist in the schema, and no schema validation rules are broken, as no other rules exist for the input elements provided in the input.

Input example that **FAILS** validation on schema above:

```
test{

    should_not_exist_one    = 21
    should_not_exist_two    = [ 22 23 24 25 ]
    invalid_rule{
        inside=5
    }

}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
Validation Error: Invalid Schema Rule: "BadRuleName" line:11 column:13

line:3 column:5 - Validation Error: /test/should_not_exist_one is not a v
alid piece of input
```

```
line:4 column:5 - Validation Error: /test/should_not_exist_two is not a v
alid piece of input
```

Notes: This input fails to validate against the provided schema because, as described above, neither `/test/should_not_exist_one` nor `/test/should_not_exist_two` exist in the schema. Also, an element exists in the input that has an invalid rule, named `BadRuleName` in the schema.

5.2.2 MinOccurs Details and Examples

The ***Minimum Occurrence*** rule describes the minimum number of times that an element must occur under its parent context. It is used mostly to denote whether a piece of input is required or optional. Most often, this rule will have a literal constant for minimum allowances. The value must be an integer. For example, `MinOccurs = 0` denotes that this element is optional under its parent context, and `MinOccurs = 1` denotes that this element is required to occur at least once under its parent. If a negative number is specified for the value of this rule, then it is treated the same as `MinOccurs = 0`. This rule may also contain a relative input lookup path from the element being validated. The syntax for this usage is `MinOccurs = "../../some/relative/input/path"`

If the lookup path describes a set containing a single value, and if that value is an integer, then that value will be used to determine the minimum allowed occurrences of the element being validated.

Schema example:

```
test{

    control{
    }
    bad_two_numbers{
    }
    bad_real{
    }
    bad_string{
    }
    valueone{
        MinOccurs=10
    }
    valuetwo{
        MinOccurs="../control"
    }
    valuethree{
        inside{
            MinOccurs=-5
        }
    }
    value_bad_one{
        inside{
            MinOccurs="../../bad_two_numbers"
        }
    }
    value_bad_two{
```

```
            inside{
                MinOccurs="../../bad_real"
            }
        }
        value_bad_three{
            inside{
                MinOccurs="../../bad_string"
            }
        }

    }
```

Input example that **PASSES** validation on schema above:

```
    test{

        control=15

        valueone=1
        valueone=2
        valueone=3
        valueone=4
        valueone=5
        valueone=6
        valueone=7
        valueone=8
        valueone=9
        valueone=10

        valuetwo=1
        valuetwo=2
        valuetwo=3
        valuetwo=4
        valuetwo=5
        valuetwo=6
        valuetwo=7
        valuetwo=8
        valuetwo=9
        valuetwo=10
        valuetwo=11
        valuetwo=12
        valuetwo=13
        valuetwo=14
        valuetwo=15

    }
```

Notes: This input passes validation against the provided schema because `valueone` must occur at least 10 times under its parent context, and it does. Also, `valuetwo` must occur at least a number of times equal to whatever integer is location at a relative location of `"../control"`. A relative lookup from

`valuetwo` to `"../control"` yields one integer with the value `15`. `valuetwo` exists under its parent context at least `15` times, so all is well.

Input examples that **FAILS** validation on schema above:

```
test{

    control=15

    valueone=1
    valueone=2
    valueone=3
    valueone=4
    valueone=5
    valueone=6
    valueone=7
    valueone=8
    valueone=9

    valuetwo=1
    valuetwo=2
    valuetwo=3
    valuetwo=4
    valuetwo=5
    valuetwo=6
    valuetwo=7
    valuetwo=8
    valuetwo=9
    valuetwo=10
    valuetwo=11
    valuetwo=12
    valuetwo=13
    valuetwo=14

}

test{
    bad_two_numbers=6
    bad_two_numbers=7
    bad_real=8.2
    bad_string='some_string'
    valuethree{
    }
    value_bad_one{
    }
    value_bad_two{
    }
    value_bad_three{
    }
}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
line:1 column:1 - Validation Error: test has 9 "valueone" occurrences - w
hen there should be a minimum occurrence of 10

line:1 column:1 - Validation Error: test has 14 "valuetwo" occurrences -
when there should be a minimum occurrence of "15" from "../control"

line:32 column:1 - Validation Error: test has 0 "valueone" occurrences -
when there should be a minimum occurrence of 10

line:37 column:5 - Validation Error: valuethree has 0 "inside" occurrence
s - when there should be a minimum occurrence of -5

line:39 column:5 - Validation Error: inside minimum occurrence checks aga
inst "../../bad_two_numbers" which returns more than one value

line:41 column:5 - Validation Error: inside minimum occurrence checks aga
inst "../../bad_real" which does not return a valid number

line:43 column:5 - Validation Error: inside minimum occurrence checks aga
inst "../../bad_string" which does not return a valid number
```

Notes: This input fails to validate against the provided schema because `valueone` only occurs 9 times under its parent context, when its `MinOccurs` rule in the schema denotes that it should occur at least 10 times. `valuetwo` should occur at least 15 times under its parent context, because its `MinOccurs` rules in the schema contains a path to `"../control"`. A relative lookup from `valuetwo` to `"../control"` yields one integer with the value `15`. However, `valuetwo` only occurs 14 times under its parent. The second `test` element in the input has zero `valueone` elements when there should be at least 10 as previously described.

5.2.3 MaxOccurs Details and Examples

The ***Maximum Occurrence*** rule describes the maximum number of times that an element is allowed to occur under its parent context. Most often, this element will have a literal constant value to describe a number of maximum allowances. The value must be integer or '**NoLimit**' (indicating that there is no upper limit on the number of times this element can occur). This rule may also have a relative input lookup path from the element being validated. If the lookup path describes a set containing a single value, and if that value is an integer, then that value will be used to determine the maximum allowed occurrences of the element being validated.

Schema example:

```
test{

    MaxOccurs=NoLimit
    control{
    }
    bad_two_numbers{
    }
```

```
        bad_real{
        }
        bad_string{
        }
        valueone{
            MaxOccurs=10
        }
        valuetwo{
            MaxOccurs="../control"
        }
        value_bad_one{
            inside{
                MinOccurs="../../bad_two_numbers"
            }
        }
        value_bad_two{
            inside{
                MinOccurs="../../bad_real"
            }
        }
        value_bad_three{
            inside{
                MinOccurs="../../bad_string"
            }
        }

    }
```

Input example that **PASSES** validation on schema above:

```
    test{

        control=15

        valueone=1
        valueone=2
        valueone=3
        valueone=4
        valueone=5
        valueone=6
        valueone=7
        valueone=8
        valueone=9
        valueone=10

        valuetwo=1
        valuetwo=2
        valuetwo=3
        valuetwo=4
        valuetwo=5
```

```
    valuetwo=6
    valuetwo=7
    valuetwo=8
    valuetwo=9
    valuetwo=10
    valuetwo=11
    valuetwo=12
    valuetwo=13
    valuetwo=14
    valuetwo=15

}
```

Input example that **FAILS** validation on schema above:

```
test{

    control=15

    valueone=1
    valueone=2
    valueone=3
    valueone=4
    valueone=5
    valueone=6
    valueone=7
    valueone=8
    valueone=9
    valueone=10
    valueone=11

    valuetwo=1
    valuetwo=2
    valuetwo=3
    valuetwo=4
    valuetwo=5
    valuetwo=6
    valuetwo=7
    valuetwo=8
    valuetwo=9
    valuetwo=10
    valuetwo=11
    valuetwo=12
    valuetwo=13
    valuetwo=14
    valuetwo=15
    valuetwo=16

}
```

```
test{
    valueone=1
    valueone=2
    valueone=3
    valueone=4
    valueone=5
    valueone=6
    valueone=7
    valueone=8
    valueone=9
    valueone=10
    valueone=11
    valueone=12
    valueone=13
    valueone=14
}
test{
    control=2
    bad_two_numbers=6
    bad_two_numbers=7
    bad_real=8.2
    bad_string='some_string'
    valueone=1
    valuetwo=1
    valuetwo=2
    valuetwo=3
    valuetwo=4
    value_bad_one{
    }
    value_bad_two{
    }
    value_bad_three{
    }
}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
line:1 column:1 - Validation Error: test has 11 "valueone" occurrences -
when there should be a maximum occurrence of 10

line:1 column:1 - Validation Error: test has 16 "valuetwo" occurrences -
when there should be a maximum occurrence of "15" from "../control"

line:36 column:1 - Validation Error: test has 14 "valueone" occurrences -
when there should be a maximum occurrence of 10

line:52 column:1 - Validation Error: test has 4 "valuetwo" occurrences -
when there should be a maximum occurrence of "2" from "../control"
```

```
line:63 column:5 - Validation Error: inside minimum occurrence checks aga
inst "../../bad_two_numbers" which returns more than one value

line:65 column:5 - Validation Error: inside minimum occurrence checks aga
inst "../../bad_real" which does not return a valid number

line:67 column:5 - Validation Error: inside minimum occurrence checks aga
inst "../../bad_string" which does not return a valid number
```

5.2.4 ValType Details and Examples

The ***Value Type*** rule checks the type of the element value in the input. This can be one of the following:

- Int - meaning a negative or positive integer
- Real - meaning a negative or positive floating point value (or integer)
- String - meaning a literal string of text

Schema example:

```
test{

    one{
        ValType=Int
    }
    two{
        ValType=Int
    }
    three{
        ValType=Int
    }
    four{
        ValType=Real
    }
    five{
        ValType=Real
    }
    six{
        ValType=Real
    }
    seven{
        ValType=String
    }
    eight{
        ValType=String
    }
    nine{
        inside{
            ValType=BadType
        }
    }
}
```

Input example that **PASSES** validation on schema above:

```
test{
    one=-8
    two=0
    three=83
    four=-9.4
    five=3
    six='+9e-3'
    seven=ThisIsAString
    eight="This Is Also A String"
}
```

Input example that **FAILS** validation on schema above:

```
test{
    one=-8.3
    two=0.3
    three="+8e-3"
    four='*'
    five=StringHere
    six='another string here'
    seven=4.5
    eight=5E-4
    nine='hello world'
}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
Validation Error: Invalid Schema Rule: Bad ValType Option "BadType" at li
ne:29 column:21 - Expected [ Int Real String ]

line:2 column:5 - Validation Error: one value "-8.3" is not of type Int

line:3 column:5 - Validation Error: two value "0.3" is not of type Int

line:4 column:5 - Validation Error: three value "+8e-3" is not of type In
t

line:5 column:5 - Validation Error: four value "*" is not of type Real

line:6 column:5 - Validation Error: five value "StringHere" is not of typ
e Real

line:7 column:5 - Validation Error: six value "another string here" is no
t of type Real
```

5.2.5 ValEnums Details and Examples

The ***Value Enumerations*** rule contains a static list of values choices. It compares the element's input value with the provided choices. If the element's value is not in the schema's list of allowed enumerations,

then this check will fail. Also, a `REF:` construct may be used to reference a SON array of values that must exist in the schema after an `EndOfSchema{}` declaration. These referenced SON arrays can be conveniently defined in one place but be used by `ValEnums` rules on many different elements. If a validation message is produced, then a short list of closest matches is provided to the user alphabetically. Note that this check is case insensitive, and if the value that is being checked is an integer, then leading zeros are ignored.

Schema example:

```
test{

    one{
        ValEnums=[ yes no maybe ]
    }
    two{
        ValEnums=[ yes no maybe ]
    }
    three{
        ValEnums=[ REF:ReferencedColors ]
    }
    four{
        ValEnums=[ REF:ReferencedNumbers ]
    }
    five{
        ValEnums=[ REF:ReferencedNumbers REF:ReferencedColors ]
    }
    six{
        ValEnums=[ REF:BadReference REF:ReferencedNumbers ]
    }

}

EndOfSchema{}

ReferencedColors=[ red orange yellow green blue indigo violet ]
ReferencedNumbers=[ 1 2 3 4 5 ]
```

Input example that **PASSES** validation on schema above:

```
test{
    one="yes"
    two='Maybe'
    three=blue
    four=4
    five=oRanGe
    five=0002
}
```

Input example that **FAILS** validation on schema above:

```
test{
    one=red
```

```
    two="Green"
    three=yes
    four=-4
    five=007
    six=something
}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
Validation Error: Invalid Schema Rule: Enum Reference "BadReference" at l
ine:19 column:20 not found in schema

line:2 column:5 - Validation Error: one value "red" is not one of the all
owed values: [ "maybe" "no" "yes" ]

line:3 column:5 - Validation Error: two value "green" is not one of the a
llowed values: [ "maybe" "no" "yes" ]

line:4 column:5 - Validation Error: three value "yes" is not one of the a
llowed values: [ ... "green" "indigo" "orange" "red" "violet" "yellow" ]

line:5 column:5 - Validation Error: four value "-4" is not one of the all
owed values: [ "1" "2" "3" "4" "5" ]

line:6 column:5 - Validation Error: five value "7" is not one of the allo
wed values: [ ... "3" "4" "5" "blue" "green" "indigo" ... ]
```

5.2.6 MinValInc Details and Examples

The ***Minimum Value Inclusive*** rule provides a number (real or integer) to which the associated input value must be greater than or equal. Most often, this rule will contain a constant number defining the minimum allowable value for this element. For example, `MinValInc = 0.0` denotes that this element's value must be zero or greater. This rule may also have a relative input lookup path from the element being validated. If the set in the input represented by the relative path is a single value, and if that value is a number, then that value will be used to determine the lowest allowed value for the element being validated. If an element at this relative lookup path exists in the input and it is not a number, then it will fail this check. However, if this element does not exist at all in the input, then this validation check is delegated to the `MinOccurs` check and will not fail.

Schema example:

```
test{

    controlone{
    }
    controltwo{
    }
    bad_two_numbers{
    }
    bad_string{
```

```
        }
        valueone{
            MinValInc=58.7
        }
        valuetwo{
            value{
                MinValInc=58.7
            }
        }
        valuethree{
            MinValInc=23
        }
        valuefour{
            value{
                MinValInc=23
            }
        }
        valuefive{
            MinValInc="../controlone"
        }
        valuesix{
            value{
                MinValInc="../../controlone"
            }
        }
        valueseven{
            MinValInc="../controltwo"
        }
        valueeight{
            value{
                MinValInc="../../controltwo"
            }
        }
        value_bad_one{
            inside{
                MinValInc="../../bad_two_numbers"
            }
        }
        value_bad_two{
            inside{
                MinValInc="../../bad_string"
            }
        }

    }
```

Input example that **PASSES** validation on schema above:

```
    test{
        controlone=15
```

```
    controltwo=-45.3
    valueone=58.7
    valuetwo=[ 65 66 67 68 58.7 ]
    valuethree=23
    valuefour=[ 38.3 30.3 23 32.34 ]
    valuefive=15
    valuesix=[ 21 22 23 24 15 ]
    valueseven=-45.3
    valueeight=[ -32.4 31.9 -30.3 -45.3 ]
}
```

Input example that **FAILS** validation on schema above:

```
test{
    controlone=15
    controltwo=-45.3
    bad_two_numbers=6
    bad_two_numbers=7
    bad_string='some_string'
    valueone=58.6
    valuetwo=[ 65 56 58.6 58 88.7 ]
    valuethree=22.9
    valuefour=[ 28.3 20.3 22.9 12.34 2e2 ]
    valuefive=14
    valuesix=[ 11 12 15 14 15.1 ]
    valueseven=-45.4
    valueeight=[ -45.4 -41.9 -100.3 -45.3 -4E-8 -7e+3 ]
    value_bad_one{
        inside=47
    }
    value_bad_two{
        inside=48
    }
    valueone='a-string'
}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
line:7 column:5 - Validation Error: valueone value "58.6" is less than th
e allowed minimum inclusive value of 58.7

line:8 column:19 - Validation Error: valuetwo value "56" is less than the
allowed minimum inclusive value of 58.7

line:8 column:22 - Validation Error: valuetwo value "58.6" is less than t
he allowed minimum inclusive value of 58.7

line:8 column:27 - Validation Error: valuetwo value "58" is less than the
allowed minimum inclusive value of 58.7
```

```
line:9 column:5 - Validation Error: valuethree value "22.9" is less than
the allowed minimum inclusive value of 23

line:10 column:22 - Validation Error: valuefour value "20.3" is less than
the allowed minimum inclusive value of 23

line:10 column:27 - Validation Error: valuefour value "22.9" is less than
the allowed minimum inclusive value of 23

line:10 column:32 - Validation Error: valuefour value "12.34" is less tha
n the allowed minimum inclusive value of 23

line:11 column:5 - Validation Error: valuefive value "14" is less than th
e allowed minimum inclusive value of "15" from "../controlone"

line:12 column:16 - Validation Error: valuesix value "11" is less than th
e allowed minimum inclusive value of "15" from "../../controlone"

line:12 column:19 - Validation Error: valuesix value "12" is less than th
e allowed minimum inclusive value of "15" from "../../controlone"

line:12 column:25 - Validation Error: valuesix value "14" is less than th
e allowed minimum inclusive value of "15" from "../../controlone"

line:13 column:5 - Validation Error: valueseven value "-45.4" is less tha
n the allowed minimum inclusive value of "-45.3" from "../controltwo"

line:14 column:18 - Validation Error: valueeight value "-45.4" is less th
an the allowed minimum inclusive value of "-45.3" from "../../controltwo"

line:14 column:30 - Validation Error: valueeight value "-100.3" is less t
han the allowed minimum inclusive value of "-45.3" from "../../controltwo
"

line:14 column:49 - Validation Error: valueeight value "-7e+3" is less th
an the allowed minimum inclusive value of "-45.3" from "../../controltwo"

line:16 column:9 - Validation Error: inside minimum inclusive value check
s against "../../bad_two_numbers" which returns more than one value

line:19 column:9 - Validation Error: inside minimum inclusive value check
s against "../../bad_string" which does not return a valid number

line:21 column:5 - Validation Error: valueone value "a-string" is wrong v
alue type for minimum inclusive value
```

5.2.7 MaxValInc Details and Examples

The ***Maximum Value Inclusive*** rule provides a number (real or integer) to which the associated input value must be less than or equal. Most often, this rule will contain a constant number defining the

maximum allowable value for this element. For example, `MaxValInc = 0.0` denotes that this element's value must be zero or less. This rule may also have a relative input lookup path from the element being validated. If the set in the input represented by the relative path is a single value, and if that value is a number, then that value will be used to determine the highest allowed value for the element being validated. If an element at this relative lookup path exists in the input and it is not a number, then it will fail this check. However, if this element does not exist at all in the input, then this validation check is delegated to the `MinOccurs` check and will not fail.

Schema example:

```
test{

    controlone{
    }
    controltwo{
    }
    bad_two_numbers{
    }
    bad_string{
    }
    valueone{
        MaxValInc=58.7
    }
    valuetwo{
        value{
            MaxValInc=58.7
        }
    }
    valuethree{
        MaxValInc=23
    }
    valuefour{
        value{
            MaxValInc=23
        }
    }
    valuefive{
        MaxValInc="../controlone"
    }
    valuesix{
        value{
            MaxValInc="../../controlone"
        }
    }
    valueseven{
        MaxValInc="../controltwo"
    }
    valueeight{
        value{
            MaxValInc="../../controltwo"
```

```
        }
    }
    value_bad_one{
        inside{
            MaxValInc="../../bad_two_numbers"
        }
    }
    value_bad_two{
        inside{
            MaxValInc="../../bad_string"
        }
    }

}
```

Input example that **PASSES** validation on schema above:

```
test{
    controlone=15
    controltwo=-45.3
    valueone=58.7
    valuetwo=[ 55 56 57 58 58.7 ]
    valuethree=23
    valuefour=[ 18.3 20.3 23 12.34 ]
    valuefive=15
    valuesix=[ 11 12 13 14 15 ]
    valueseven=-45.3
    valueeight=[ -52.4 -51.9 -100.3 -45.3 ]
}
```

Input example that **FAILS** validation on schema above:

```
test{
    controlone=15
    controltwo=-45.3
    bad_two_numbers=6
    bad_two_numbers=7
    bad_string='some_string'
    valueone=58.8
    valuetwo=[ 65 56 58.8 58 88.7 ]
    valuethree=23.9
    valuefour=[ 18.3 20.3 23.1 12.34 2e2 ]
    valuefive=19
    valuesix=[ 11 12 18.2 14 15.1 ]
    valueseven=-45.1
    valueeight=[ -52.4 -41.9 -100.3 -45.3 -4E-8 -7e+3 ]
    value_bad_one{
        inside=47
    }
    value_bad_two{
        inside=48
```

```
    }
    valueone='a-string'
}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
line:7 column:5 - Validation Error: valueone value "58.8" is greater than
the allowed maximum inclusive value of 58.7

line:8 column:16 - Validation Error: valuetwo value "65" is greater than
the allowed maximum inclusive value of 58.7

line:8 column:22 - Validation Error: valuetwo value "58.8" is greater tha
n the allowed maximum inclusive value of 58.7

line:8 column:30 - Validation Error: valuetwo value "88.7" is greater tha
n the allowed maximum inclusive value of 58.7

line:9 column:5 - Validation Error: valuethree value "23.9" is greater th
an the allowed maximum inclusive value of 23

line:10 column:27 - Validation Error: valuefour value "23.1" is greater t
han the allowed maximum inclusive value of 23

line:10 column:38 - Validation Error: valuefour value "2e2" is greater th
an the allowed maximum inclusive value of 23

line:11 column:5 - Validation Error: valuefive value "19" is greater than
the allowed maximum inclusive value of "15" from "../controlone"

line:12 column:22 - Validation Error: valuesix value "18.2" is greater th
an the allowed maximum inclusive value of "15" from "../../controlone"

line:12 column:30 - Validation Error: valuesix value "15.1" is greater th
an the allowed maximum inclusive value of "15" from "../../controlone"

line:13 column:5 - Validation Error: valueseven value "-45.1" is greater
than the allowed maximum inclusive value of "-45.3" from "../controltwo"

line:14 column:24 - Validation Error: valueeight value "-41.9" is greater
than the allowed maximum inclusive value of "-45.3" from "../../controltw
o"

line:14 column:43 - Validation Error: valueeight value "-4E-8" is greater
than the allowed maximum inclusive value of "-45.3" from "../../controltw
o"

line:16 column:9 - Validation Error: inside maximum inclusive value check
s against "../../bad_two_numbers" which returns more than one value
```

```
line:19 column:9 - Validation Error: inside maximum inclusive value check
s against "../../bad_string" which does not return a valid number

line:21 column:5 - Validation Error: valueone value "a-string" is wrong v
alue type for maximum inclusive value
```

5.2.8 MinValExc Details and Examples

The ***Minimum Value Exclusive*** rule provides a number (real or integer) to which the associated input value must be greater. Most often, this rule will contain a constant number, and the associated input value must be greater than this number. For example, `MinValExc = 0.0` denotes that this element value must be greater than zero (not equal). This rule may also have a relative input lookup path from the element being validated. If the set in the input represented by the relative path is a single value, and if that value is a number, then that value will be used to determine the minimum exclusive allowed input value. If an element at this relative lookup path exists in the input and it is not a number, then it will fail this check. However, if this element does not exist at all in the input, then this validation check is delegated to the `MinOccurs` check and will not fail.

Schema example:

```
test{

    controlone{
    }
    controltwo{
    }
    bad_two_numbers{
    }
    bad_string{
    }
    valueone{
        MinValExc=58.7
    }
    valuetwo{
        value{
            MinValExc=58.7
        }
    }
    valuethree{
        MinValExc=23
    }
    valuefour{
        value{
            MinValExc=23
        }
    }
    valuefive{
        MinValExc="../controlone"
    }
    valuesix{
```

```
        value{
            MinValExc="../../controlone"
        }
    }
    valueseven{
        MinValExc="../controltwo"
    }
    valueeight{
        value{
            MinValExc="../../controltwo"
        }
    }
    valuenine{
        MinValExc=NoLimit
    }
    value_bad_one{
        inside{
            MinValExc="../../bad_two_numbers"
        }
    }
    value_bad_two{
        inside{
            MinValExc="../../bad_string"
        }
    }

}
```

Input example that **PASSES** validation on schema above:

```
test{
    controlone=15
    controltwo=-45.3
    valueone=58.8
    valuetwo=[ 65 66 67 68 58.8 ]
    valuethree=23.1
    valuefour=[ 38.3 30.3 23.1 32.34 ]
    valuefive=16
    valuesix=[ 21 22 23 24 16 ]
    valueseven=-45.2
    valueeight=[ -32.4 31.9 -30.3 -45.2 ]
    valuenine=-2000.90
}
```

Input example that **FAILS** validation on schema above:

```
test{
    controlone=15
    controltwo=-453E-1
    bad_two_numbers=6
    bad_two_numbers=7
```

```
    bad_string='some_string'
    valueone=58.7
    valuetwo=[ 65E-1 66 7 68 58.7 ]
    valuethree=23
    valuefour=[ 383E-2 3.3 23 32.34 ]
    valuefive=15
    valuesix=[ -21 22 2.3E-4 24 15 ]
    valueseven=-45.3
    valueeight=[ -132.4 -3.19E5 -30.3 -45.3 ]
    valuenine=-2000.90
    value_bad_one{
        inside=47
    }
    value_bad_two{
        inside=48
    }
    valueone='a-string'
}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
line:7 column:5 - Validation Error: valueone value "58.7" is less than or
equal to the allowed minimum exclusive value of 58.7

line:8 column:16 - Validation Error: valuetwo value "65E-1" is less than
or equal to the allowed minimum exclusive value of 58.7

line:8 column:25 - Validation Error: valuetwo value "7" is less than or e
qual to the allowed minimum exclusive value of 58.7

line:8 column:30 - Validation Error: valuetwo value "58.7" is less than o
r equal to the allowed minimum exclusive value of 58.7

line:9 column:5 - Validation Error: valuethree value "23" is less than or
equal to the allowed minimum exclusive value of 23

line:10 column:17 - Validation Error: valuefour value "383E-2" is less th
an or equal to the allowed minimum exclusive value of 23

line:10 column:24 - Validation Error: valuefour value "3.3" is less than
or equal to the allowed minimum exclusive value of 23

line:10 column:28 - Validation Error: valuefour value "23" is less than o
r equal to the allowed minimum exclusive value of 23

line:11 column:5 - Validation Error: valuefive value "15" is less than or
equal to the allowed minimum exclusive value of "15" from "../controlone"

line:12 column:16 - Validation Error: valuesix value "-21" is less than o
```

```
r equal to the allowed minimum exclusive value of "15" from "../../contro
lone"

line:12 column:23 - Validation Error: valuesix value "2.3E-4" is less tha
n or equal to the allowed minimum exclusive value of "15" from "../../con
trolone"

line:12 column:33 - Validation Error: valuesix value "15" is less than or
equal to the allowed minimum exclusive value of "15" from "../../controlo
ne"

line:13 column:5 - Validation Error: valueseven value "-45.3" is less tha
n or equal to the allowed minimum exclusive value of "-453E-1" from "../c
ontroltwo"

line:14 column:18 - Validation Error: valueeight value "-132.4" is less t
han or equal to the allowed minimum exclusive value of "-453E-1" from ".
./../controltwo"

line:14 column:25 - Validation Error: valueeight value "-3.19E5" is less
than or equal to the allowed minimum exclusive value of "-453E-1" from ".
./../controltwo"

line:14 column:39 - Validation Error: valueeight value "-45.3" is less th
an or equal to the allowed minimum exclusive value of "-453E-1" from "../
../controltwo"

line:17 column:9 - Validation Error: inside minimum exclusive value check
s against "../../bad_two_numbers" which returns more than one value

line:20 column:9 - Validation Error: inside minimum exclusive value check
s against "../../bad_string" which does not return a valid number

line:22 column:5 - Validation Error: valueone value "a-string" is wrong v
alue type for minimum exclusive value
```

5.2.9 MaxValExc Details and Examples

The ***Maximum Value Exclusive*** rule provides a number (real or integer) to which the associated input value must be less. Most often, this rule will contain a constant number, and the associated input value must be less than this number. For example, `MaxValExc = 0.0` denotes that this element value must be less than zero (not equal). This rule may also have a relative input lookup path from the element being validated. If the set in the input represented by the relative path is a single value, and if that value is a number, then that value will be used to determine the maximum exclusive allowed input value. If an element at this relative lookup path exists in the input and it is not a number, then it will fail this check. However, if this element does not exist at all in the input, then this validation check is delegated to the `MinOccurs` check and will not fail.

Schema example:

```
test{

    controlone{
    }
    controltwo{
    }
    bad_two_numbers{
    }
    bad_string{
    }
    valueone{
        MaxValExc=58.7
    }
    valuetwo{
        value{
            MaxValExc=58.7
        }
    }
    valuethree{
        MaxValExc=23
    }
    valuefour{
        value{
            MaxValExc=23
        }
    }
    valuefive{
        MaxValExc="../controlone"
    }
    valuesix{
        value{
            MaxValExc="../../controlone"
        }
    }
    valueseven{
        MaxValExc="../controltwo"
    }
    valueeight{
        value{
            MaxValExc="../../controltwo"
        }
    }
    valuenine{
        MaxValExc=NoLimit
    }
    value_bad_one{
        inside{
            MaxValExc="../../bad_two_numbers"
        }
    }
```

```
    value_bad_two{
        inside{
            MaxValExc="../../bad_string"
        }
    }

}
```

Input example that **PASSES** validation on schema above:

```
test{
    controlone=15.1
    controltwo=-452E-1
    valueone=58.6
    valuetwo=[ 55 56 57 58 58.6 ]
    valuethree=22.9
    valuefour=[ 18.3 20.3 22.9 12.34 ]
    valuefive=15
    valuesix=[ 11 12 13 14 15 ]
    valueseven=-45.3
    valueeight=[ -52.4 -51.9 -100.3 -45.3 ]
    valuenine=2000.90
}
```

Input example that **FAILS** validation on schema above:

```
test{
    controlone=15
    controltwo=-453e-1
    bad_two_numbers=6
    bad_two_numbers=7
    bad_string='some_string'
    valueone=58.7
    valuetwo=[ 65 59 57 58 58.7 ]
    valuethree=23
    valuefour=[ 18.3 29.3 23 12.34 ]
    valuefive=15
    valuesix=[ 11 12 13 14 15 17.3 ]
    valueseven=-45.3
    valueeight=[ -52.4 -51.9 -10.3 -45.3 ]
    valuenine=2000.90
    value_bad_one{
        inside=47
    }
    value_bad_two{
        inside=48
    }
    valueone='a-string'
}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
line:7 column:5 - Validation Error: valueone value "58.7" is greater than
or equal to the allowed maximum exclusive value of 58.7

line:8 column:16 - Validation Error: valuetwo value "65" is greater than
or equal to the allowed maximum exclusive value of 58.7

line:8 column:19 - Validation Error: valuetwo value "59" is greater than
or equal to the allowed maximum exclusive value of 58.7

line:8 column:28 - Validation Error: valuetwo value "58.7" is greater tha
n or equal to the allowed maximum exclusive value of 58.7

line:9 column:5 - Validation Error: valuethree value "23" is greater than
or equal to the allowed maximum exclusive value of 23

line:10 column:22 - Validation Error: valuefour value "29.3" is greater t
han or equal to the allowed maximum exclusive value of 23

line:10 column:27 - Validation Error: valuefour value "23" is greater tha
n or equal to the allowed maximum exclusive value of 23

line:11 column:5 - Validation Error: valuefive value "15" is greater than
or equal to the allowed maximum exclusive value of "15" from "../controlo
ne"

line:12 column:28 - Validation Error: valuesix value "15" is greater than
or equal to the allowed maximum exclusive value of "15" from "../../contr
olone"

line:12 column:31 - Validation Error: valuesix value "17.3" is greater th
an or equal to the allowed maximum exclusive value of "15" from "../../co
ntrolone"

line:13 column:5 - Validation Error: valueseven value "-45.3" is greater
than or equal to the allowed maximum exclusive value of "-453e-1" from ".
./controltwo"

line:14 column:30 - Validation Error: valueeight value "-10.3" is greater
than or equal to the allowed maximum exclusive value of "-453e-1" from ".
./../controltwo"

line:14 column:36 - Validation Error: valueeight value "-45.3" is greater
than or equal to the allowed maximum exclusive value of "-453e-1" from ".
./../controltwo"

line:17 column:9 - Validation Error: inside maximum exclusive value check
s against "../../bad_two_numbers" which returns more than one value
```

```
line:20 column:9 - Validation Error: inside maximum exclusive value check
s against "../../bad_string" which does not return a valid number

line:22 column:5 - Validation Error: valueone value "a-string" is wrong v
alue type for maximum exclusive value
```

5.2.10 ExistsIn Details and Examples

The ***Exists In*** rule is used as a key to stipulate that an element in the input must be defined somewhere else in the input. This rule will always contain one or more relative input lookup paths from the element being validated. The pieces of input at these paths will be collected into a set. This rule may also contain one or more optional constant values. If these exist, then the constant values will also be added to the set. Then, all of the values in the input being validated by this rule must exist in the set built from the lookup paths and the constant values in order to pass the validation. If any element does not exist in this set, then the validation check fails. This rule may use an optional `Abs` modifier flag that can occur as a parenthetical identifier. The `Abs` modifier flag indicates that the absolute values of all numbers added to the set checked for existence are used. Then, even if the value of the element being validated is negative and a value at one of the rule's relative input lookup paths is positive, but they have the same absolute value, this validation check will pass. `EXTRA:` may be used within an `ExistsIn` to specify constant values that are allowed. An `EXTRAREF:` construct may be used to reference a SON array of values that must exist in the schema after an `EndOfSchema{}` declaration. The values are also allowed by the `ExistsIn` rule. These referenced SON arrays can be conveniently defined in one place but be used by `ExistsIn` rules on many different elements. If the allowed `EXTRA` values are actually a contiguous range of integer values, then a `RANGE:[ start end ]` construct may be used for convenience instead of writing a separate `EXTRA:` for every element. These are all shown in the syntax example below. Note that this check is case insensitive, and if the value that is being checked is an integer, then leading zeros are ignored.

Schema example:

```
test{

    defineone{
    }
    definetwo{
    }
    definethree{
    }
    useone{
        value{
            ExistsIn=[ "../../defineone/value"
                       "../../definetwo/value"
                       "../../definethree/value" ]
        }
    }
    usetwo{
        value{
            ExistsIn=[ EXTRA:"ford"
                       EXTRA:"chevy"
                       EXTRA:"bmw"
```

```
                    EXTRAREF:ReferencedColors
                    EXTRAREF:ReferencedNumbers
                    "../../defineone/value"
                    "../../definetwo/value"
                    "../../definethree/value" ]
    }
}
usethree{
    value{
        ExistsIn(Abs)=[ EXTRAREF:ReferencedNumbers
                        "../../definetwo/value" ]
    }
}
usefour_reg{
    value{
        ExistsIn=[ EXTRA:"hamburger"
                   EXTRA:"hotdog"
                   EXTRA:800
                   EXTRA:900
                   "../../defineone/value"
                   "../../definethree/value" ]
    }
}
usefive_reg{
    value{
        ExistsIn=[ EXTRAREF:ReferencedColors
                   "../../definetwo/value"
                   "../../definethree/value"
                    EXTRA:0
                    RANGE:[ 1200 1300 ]
                    RANGE:[ 1400 1500 ]
                    RANGE:[ 1600 1700 ]
                    RANGE:[ 1800 1900 ] ]
    }
}
usefour_abs{
    value{
        ExistsIn(Abs)=[ EXTRA:"hamburger"
                   EXTRA:"hotdog"
                   EXTRA:800
                   EXTRA:900
                   "../../defineone/value"
                   "../../definethree/value" ]
    }
}
usefive_abs{
    value{
        ExistsIn(Abs)=[ EXTRAREF:ReferencedColors
                   "../../definetwo/value"
                   "../../definethree/value"
```

```
                        EXTRA:0
                        RANGE:[ 1200 1300 ]
                        RANGE:[ 1400 1500 ]
                        RANGE:[ 1600 1700 ]
                        RANGE:[ 1800 1900 ] ]
        }
    }
    usesix{
        value{
            ExistsIn(BadFlag)=[ "../../defineone/value" ]
        }
    }
    useseven{
        value{
            ExistsIn=[ "../../defineone/value"
                       RANGE:[ 25 fifty ] ]
        }
    }
    useeight{
        value{
            ExistsIn=[ "../../defineone/value"
                       RANGE:[ 50 25 ] ]
        }
    }
    usenine{
        value{
            ExistsIn=[ "../../defineone/value"
                       RANGE:[ 25 50 100 ] ]
        }
    }
    useten{
        value{
            ExistsIn=[ EXTRAREF:BadReference
                       "../../defineone/value" ]
        }
    }
    useeleven{
        value{
            ExistsIn=[ "../../../../defineone/value" ]
        }
    }

}
EndOfSchema{}
ReferencedColors=[ red orange yellow green blue indigo violet ]
ReferencedNumbers=[ 1 2 3 4 5 ]
```

Input example that **PASSES** validation on schema above:

```
test{

    defineone=one
    defineone=two
    defineone=three
    defineone=four

    definetwo=[ england spain germany italy canada ]
    definetwo=-200
    definetwo=300
    definetwo=[ 500 -600 ]

    definethree=science
    definethree=math
    definethree=[ geography economics recess lunch ]

    useone=two
    useone=germany
    useone=[ three recess lunch italy canada ]

    usetwo=[ ford bmw red 1 4 math ]
    usetwo=3
    usetwo=blue

    usethree=england
    usethree=italy
    usethree=[ 5 "3" -2 canada "1" ]
    usethree=-4

    usefour_abs=geography
    usefour_abs=[ hamburger 900 math hotdog "four" ]
    usefour_abs=three
    usefour_abs=[ 800 -800 ]
    usefour_abs=-900

    usefive_abs=orange
    usefive_abs=economics
    usefive_abs=[ "indigo" violet "geography" ]
    usefive_abs=science
    usefive_abs=[ 600 -600 300 -300 1200 1300 ]
    usefive_abs=200
    usefive_abs=[ -500 500 -200 -1850 ]
    usefive_abs=-1675

}
```

Input example that **FAILS** validation on schema above:

```
test{
```

```
    defineone=one
    defineone=two
    defineone=three
    defineone=four

    definetwo=[ england spain germany italy canada ]
    definetwo=-200
    definetwo=300
    definetwo=[ 500 -600 ]

    definethree=science
    definethree=math
    definethree=[ geography economics recess lunch ]

    useone=seven
    useone=japan
    useone=[ three spelling yellow italy canada 2 ]

    usetwo=[ ford honda red -1 4 math ]
    usetwo=-3
    usetwo=purple

    usethree=red
    usethree=three
    usethree=[ 5 "3" -2.3 blue "1" ]
    usethree=lunch

    usefour_reg=geography
    usefour_reg=[ hamburger 900 spain hotdog fries ]
    usefour_reg=orange
    usefour_reg=[ 800 -800 ]
    usefour_reg=-900

    usefive_reg=orange
    usefive_reg=economics
    usefive_reg=[ "indigo" violet "geography" ]
    usefive_reg=science
    usefive_reg=[ 600 2 300 five ]
    usefive_reg=200
    usefive_reg=[ -500 -3 -200 ]

    usesix=one
    useseven=two
    useeight=three
    usenine=four
    useten=one
    useeleven=two

}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
Validation Error: Invalid Schema Rule: "50" start of range is greater tha
n or equal to "25" end of range at line:92 column:32

Validation Error: Invalid Schema Rule: Bad ExistsIn Option "BadFlag" at l
ine:80 column:22 - Expected [ Abs ]

Validation Error: Invalid Schema Rule: Bad ExistsIn Path "../../../../def
ineone/value" at line:109 column:24

Validation Error: Invalid Schema Rule: Enum Reference "BadReference" at l
ine:103 column:24 not found in schema

Validation Error: Invalid Schema Rule: Range does not have exactly two va
lues at line:98 column:24

Validation Error: Invalid Schema Rule: fifty range value not a valid numb
er at line:86 column:35

line:17 column:12 - Validation Error: useone value "seven" does not exist
in set: [ ../../defineone/value ../../definetwo/value ../../definethree/v
alue ]

line:18 column:12 - Validation Error: useone value "japan" does not exist
in set: [ ../../defineone/value ../../definetwo/value ../../definethree/v
alue ]

line:19 column:20 - Validation Error: useone value "spelling" does not ex
ist in set: [ ../../defineone/value ../../definetwo/value ../../definethr
ee/value ]

line:19 column:29 - Validation Error: useone value "yellow" does not exis
t in set: [ ../../defineone/value ../../definetwo/value ../../definethree
/value ]

line:19 column:49 - Validation Error: useone value "2" does not exist in
set: [ ../../defineone/value ../../definetwo/value ../../definethree/valu
e ]

line:21 column:19 - Validation Error: usetwo value "honda" does not exist
in set: [ ../../defineone/value ../../definetwo/value ../../definethree/v
alue ]

line:21 column:29 - Validation Error: usetwo value "-1" does not exist in
set: [ ../../defineone/value ../../definetwo/value ../../definethree/valu
e ]

line:22 column:12 - Validation Error: usetwo value "-3" does not exist in
```

```
set: [ ../../defineone/value ../../definetwo/value ../../definethree/valu
e ]

line:23 column:12 - Validation Error: usetwo value "purple" does not exis
t in set: [ ../../defineone/value ../../definetwo/value ../../definethree
/value ]

line:25 column:14 - Validation Error: usethree value "red" does not exist
in set: [ ../../definetwo/value ]

line:26 column:14 - Validation Error: usethree value "three" does not exi
st in set: [ ../../definetwo/value ]

line:27 column:22 - Validation Error: usethree value "2.3" does not exist
in set: [ ../../definetwo/value ]

line:27 column:27 - Validation Error: usethree value "blue" does not exis
t in set: [ ../../definetwo/value ]

line:28 column:14 - Validation Error: usethree value "lunch" does not exi
st in set: [ ../../definetwo/value ]

line:31 column:33 - Validation Error: usefour_reg value "spain" does not
exist in set: [ ../../defineone/value ../../definethree/value ]

line:31 column:46 - Validation Error: usefour_reg value "fries" does not
exist in set: [ ../../defineone/value ../../definethree/value ]

line:32 column:17 - Validation Error: usefour_reg value "orange" does not
exist in set: [ ../../defineone/value ../../definethree/value ]

line:33 column:23 - Validation Error: usefour_reg value "-800" does not e
xist in set: [ ../../defineone/value ../../definethree/value ]

line:34 column:17 - Validation Error: usefour_reg value "-900" does not e
xist in set: [ ../../defineone/value ../../definethree/value ]

line:40 column:19 - Validation Error: usefive_reg value "600" does not ex
ist in set: [ ../../definetwo/value ./../definethree/value ]

line:40 column:23 - Validation Error: usefive_reg value "2" does not exis
t in set: [ ../../definetwo/value ./../definethree/value ]

line:40 column:29 - Validation Error: usefive_reg value "five" does not e
xist in set: [ ../../definetwo/value ../../definethree/value ]

line:41 column:17 - Validation Error: usefive_reg value "200" does not ex
ist in set: [ ../../definetwo/value ../../definethree/value ]

line:42 column:19 - Validation Error: usefive_reg value "-500" does not e
```

```
xist in set: [ ../../definetwo/value ../../definethree/value ]

line:42 column:24 - Validation Error: usefive_reg value "-3" does not exi
st in set: [ ../../definetwo/value ../../definethree/value ]
```

5.2.11 NotExistsIn Details and Examples

The ***Not Exists In*** rule will always contain one or more relative input lookup paths from the element being validated. The pieces of input at these paths will be collected into a set. If the value of the element being validated exists in this set, then this validation check fails. If it does not exist, then the validation check passes. This rule may use an optional `Abs` modifier flag that can occur as a parenthetical identifier. The `Abs` modifier flag indicates that the absolute value of all numbers added to the set checked for existence are used. Then, even if the value of the element being validated is negative and a value at one of the rule's relative input lookup paths is positive, but they have the same absolute value, then this validation check will fail. Note that this check is case insensitive, and if the value that is being checked is an integer, then leading zeros are ignored.

Schema example:

```
test{

    defineone{
    }
    definetwo{
    }
    definethree{
    }
    useone{
        value{
            NotExistsIn=[ "../../defineone/value"
                          "../../definetwo/value"
                          "../../definethree/value" ]
        }
    }
    usetwo{
        value{
            NotExistsIn(Abs)=[ "../../defineone/value"
                               "../../definetwo/value" ]
        }
    }
    usethree{
        value{
            NotExistsIn=[ "../../defineone/value"
                          "../../definethree/value" ]
        }
    }
    usefour{
        value{
            NotExistsIn=[ "../../definetwo/value"
                          "../../definethree/value" ]
        }
```

```
        }
        usefive{
            value{
                NotExistsIn=[ "../../definethree/value" ]
            }
        }
        usesix{
            value{
                NotExistsIn(BadFlag)=[ "../../defineone/value" ]
            }
        }

    }
```

Input example that **PASSES** validation on schema above:

```
    test{

        defineone=one
        defineone=two
        defineone=three
        defineone=four
        defineone=0

        definetwo=[ england spain germany italy canada ]
        definetwo=-200
        definetwo=300
        definetwo=[ 500 0 -600 ]

        definethree=science
        definethree=math
        definethree=[ geography economics 0 recess lunch ]

        useone=200
        useone=japan
        useone=[ -500 600 seven -300 art ]

        usetwo=[ science "recess" ]
        usetwo="lunch"
        usetwo=economics
        usetwo=[ "math" geography ]

        usethree=canada
        usethree=england
        usethree=[ -200 "italy" 300 ]
        usethree="-600"

        usefour="one"
        usefour=[ "two" one ]
        usefour="four"
```

```
    usefour=[ "four" three ]
    usefour="three"

    usefive=[ 300 -600 ]
    usefive="one"
    usefive=[ three italy "england" ]

}
```

Input example that **FAILS** validation on schema above:

```
test{

    defineone=one
    defineone=two
    defineone=three
    defineone=four

    definetwo=[ england spain germany italy canada ]
    definetwo=-200
    definetwo=300
    definetwo=[ 500 0 -600 ]

    definethree=science
    definethree=math
    definethree=[ geography economics 0 recess lunch ]

    useone=two
    useone=germany
    useone=[ three recess lunch italy canada ]

    usetwo=[ two germany -600 ]
    usetwo="four"
    usetwo="600"
    usetwo=[ -200 200 one ]

    usethree=four
    usethree=lunch
    usethree=[ two "three" ]
    usethree="science"

    usefour=300
    usefour=[ -600 economics ]
    usefour=recess
    usefour=[ lunch -200 ]
    usefour=math

    usefive=[ recess "math" ]
    usefive="science"
    usefive=[ math economics "geography" ]
```

```
    usesix=one

}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
Validation Error: Invalid Schema Rule: Bad NotExistsIn Option "BadFlag" a
t line:41 column:25 - Expected [ Abs ]

line:17 column:12 - Validation Error: useone value "two" also exists at "
../../defineone/value" on line:4 column:15

line:18 column:12 - Validation Error: useone value "germany" also exists
at "../../definetwo/value" on line:8 column:31

line:19 column:14 - Validation Error: useone value "three" also exists at
"../../defineone/value" on line:5 column:15

line:19 column:20 - Validation Error: useone value "recess" also exists a
t "../../definethree/value" on line:15 column:41

line:19 column:27 - Validation Error: useone value "lunch" also exists at
"../../definethree/value" on line:15 column:48

line:19 column:33 - Validation Error: useone value "italy" also exists at
"../../definetwo/value" on line:8 column:39

line:19 column:39 - Validation Error: useone value "canada" also exists a
t "../../definetwo/value" on line:8 column:45

line:21 column:14 - Validation Error: usetwo value "two" also exists at "
../../defineone/value" on line:4 column:15

line:21 column:18 - Validation Error: usetwo value "germany" also exists
at "../../definetwo/value" on line:8 column:31

line:21 column:26 - Validation Error: usetwo value "600" also exists at "
../../definetwo/value" on line:11 column:23

line:22 column:12 - Validation Error: usetwo value "four" also exists at
"../../defineone/value" on line:6 column:15

line:23 column:12 - Validation Error: usetwo value "600" also exists at "
../../definetwo/value" on line:11 column:23

line:24 column:14 - Validation Error: usetwo value "200" also exists at "
../../definetwo/value" on line:9 column:15
```

```
line:24 column:19 - Validation Error: usetwo value "200" also exists at "
../../definetwo/value" on line:9 column:15

line:24 column:23 - Validation Error: usetwo value "one" also exists at "
../../defineone/value" on line:3 column:15

line:26 column:14 - Validation Error: usethree value "four" also exists a
t "../../defineone/value" on line:6 column:15

line:27 column:14 - Validation Error: usethree value "lunch" also exists
at "../../definethree/value" on line:15 column:48

line:28 column:16 - Validation Error: usethree value "two" also exists at
"../../defineone/value" on line:4 column:15

line:28 column:20 - Validation Error: usethree value "three" also exists
at "../../defineone/value" on line:5 column:15

line:29 column:14 - Validation Error: usethree value "science" also exist
s at "../../definethree/value" on line:13 column:17

line:31 column:13 - Validation Error: usefour value "300" also exists at
"../../definetwo/value" on line:10 column:15

line:32 column:15 - Validation Error: usefour value "-600" also exists at
"../../definetwo/value" on line:11 column:23

line:32 column:20 - Validation Error: usefour value "economics" also exis
ts at "../../definethree/value" on line:15 column:29

line:33 column:13 - Validation Error: usefour value "recess" also exists
at "../../definethree/value" on line:15 column:41

line:34 column:15 - Validation Error: usefour value "lunch" also exists a
t "../../definethree/value" on line:15 column:48

line:34 column:21 - Validation Error: usefour value "-200" also exists at
"../../definetwo/value" on line:9 column:15

line:35 column:13 - Validation Error: usefour value "math" also exists at
"../../definethree/value" on line:14 column:17

line:37 column:15 - Validation Error: usefive value "recess" also exists
at "../../definethree/value" on line:15 column:41

line:37 column:22 - Validation Error: usefive value "math" also exists at
"../../definethree/value" on line:14 column:17

line:38 column:13 - Validation Error: usefive value "science" also exists
at "../../definethree/value" on line:13 column:17
```

```
line:39 column:15 - Validation Error: usefive value "math" also exists at
"../../definethree/value" on line:14 column:17

line:39 column:20 - Validation Error: usefive value "economics" also exis
ts at "../../definethree/value" on line:15 column:29

line:39 column:30 - Validation Error: usefive value "geography" also exis
ts at "../../definethree/value" on line:15 column:19
```

5.2.12 SumOver Details and Examples

The ***Sum Over*** rule must always contain a **context expression** and an **expected sum value**. The expected sum value is the desired sum when all of the elements in the given context are summed. The context contains a relative ancestry path in the input hierarchy that the values will be summed over. For a simple array, this will usually be "**..**" but may go back further in lineage if needed (e.g., "**../../..**").

Schema example:

```
test{

    container{
        inside{
            SumOver("../..")=107.6
        }
    }
    array{
        value{
            SumOver("..")=209.4
        }
    }
    invalid_array{
        value{
            SumOver("..")=123.4
        }
    }

}
```

Input example that **PASSES** validation on schema above:

```
test{

    container{
        inside=59.4
    }
    container{
        inside=24.9
    }
    container{
        inside=23.3
```

```
    }
    array=[ 4.5 87.3 83.2 34.4 ]

}
```

Input example that **FAILS** validation on schema above:

```
test{

    container{
        inside=59.4
    }
    container{
        inside=28.9
    }
    container{
        inside=23.3
    }
    array=[ 4.5 87.3 83.5 34.4 ]
    invalid_array= [ 1.2 4.5 something 8.8 ]

}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
line:1 column:1 - Validation Error: test children "inside" sum to 111.6 -
instead of the required sum of 107.6

line:12 column:5 - Validation Error: array children "value" sum to 209.7
- instead of the required sum of 209.4

line:13 column:30 - Validation Error: invalid_array value "something" is
wrong value type for sum over
```

5.2.13 SumOverGroup Details and Examples

The ***Sum Over Group*** rule must always contain a **context path**, a **group sum value**, a **compare path**, and a **group divide value**. The compare path is used to acquire another element in the input hierarchy relative to the current element being validated. This value must exist in the input and be a number. Then, this value is divided by the group divide value. This performs integer division to split the input element that will be added into groups. Then each group must successfully add to the group sum value. If any group does not add to the group sum value, then this validation check fails. If every group (when split by performing an integer division on the value at the compare path relative location by the group divide value) adds to the same desired group sum, then this validation check passes.

Schema example:

```
test{
    inside{
        id{
        }
```

```
            container{
                id{
                }
                inside{
                    SumOverGroup("../..")=[ ComparePath="../id"
                                            GroupDivide=1000
                                            GroupSum=107.6 ]
                }
                badoptions{
                    badruleone{
                        SumOverGroup("../..")=[
                                             GroupDivide=1000
                                             GroupSum=107.6 ]
                    }
                    badruletwo{
                        SumOverGroup("../..")=[ ComparePath="../id"

                                             GroupSum=107.6 ]
                    }
                    badrulethree{
                        SumOverGroup("../..")=[ ComparePath="../id"
                                             GroupDivide=1000
                                            ]
                    }
                }
            }
            array{
                value{
                    SumOverGroup("../../..")=[ ComparePath="../../id"
                                            GroupDivide=10
                                            GroupSum=418.8 ]
                }
            }
            invalid_array{
                value{
                    SumOverGroup("../../..")=[ ComparePath="../../id"
                                            GroupDivide=100
                                            GroupSum=123.4 ]
                }
            }
        }
    }
```

Input example that **PASSES** validation on schema above:

```
    test{

        inside{

            id=121
```

```
    container{
        id=72123
        inside=59.4
    }
    container{
        id=72456
        inside=24.9
    }
    container{
        id=72789
        inside=23.3
    }
    container{
        id=82123
        inside=59.6
    }
    container{
        id=82456
        inside=44.7
    }
    container{
        id=82789
        inside=3.3
    }
    container{
        id=92123
        inside=0.4
    }
    container{
        id=92456
        inside=107.1
    }
    container{
        id=92789
        inside=0.1
    }

    array=[ 4.5 87.3 83.2 54.4 ]

}

inside{
    id=124
    array=[ 4.5 67.3 83.2 34.4 ]
}

inside{
    id=1324
    array=[ 4.5 87.3 83.2 14.4 ]
```

```
        }

        inside{
            id=1322
            array=[ 24.5 87.3 83.2 34.4 ]
        }

    }
```

Input example that **FAILS** validation on schema above:

```
    test{

        inside{

            id=121

            container{
                id=72123
                inside=59.4
            }
            container{
                id=72456
                inside=14.9
            }
            container{
                id=72789
                inside=23.3
            }
            container{
                id=82123
                inside=59.6
            }
            container{
                id=82456
                inside=54.7
            }
            container{
                id=82789
                inside=83.3
            }
            container{
                id=92123
                inside=9.4
            }
            container{
                id=92456
                inside=107.1
            }
            container{
```

```
            id=92789
            inside=0.8
            badoptions{
            }
        }

        array=[ 4.9 87.3 3.2 54.4 ]

    }

    inside{
        id=124
        array=[ 4.5 67.3 83.2 134.4 ]
    }

    inside{
        id=1324
        array=[ 4.5 97.3 83.2 14.1 ]
    }

    inside{
        id=1322
        array=[ 24.5 87.3 83.2 14.4 ]
        invalid_array= [ 1.2 4.5 something 8.8 ]
    }

}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
Validation Error: Invalid Schema Rule: SumOverGroup missing ComparePath a
t line:15 column:21

Validation Error: Invalid Schema Rule: SumOverGroup missing GroupDivide a
t line:20 column:21

Validation Error: Invalid Schema Rule: SumOverGroup missing GroupSum at l
ine:25 column:21

line:1 column:1 - Validation Error: test children "value" sum to 408.5 fo
r 1320 group - instead of the required sum of 418.8

line:1 column:1 - Validation Error: test children "value" sum to 439.2 fo
r 120 group - instead of the required sum of 418.8

line:3 column:5 - Validation Error: inside children "inside" sum to 97.6
for 72000 group - instead of the required sum of 107.6

line:3 column:5 - Validation Error: inside children "inside" sum to 117.3
```

```
for 92000 group - instead of the required sum of 107.6

line:3 column:5 - Validation Error: inside children "inside" sum to 197.6
for 82000 group - instead of the required sum of 107.6

line:63 column:34 - Validation Error: invalid_array value "something" is
wrong value type for sum over group
```

5.2.14 IncreaseOver Details and Examples

The ***Increase Over*** rule must contain a required modifier flag that occurs as a parenthetical identifier and indicates the monotonicity. The flag must either be `Strict`, meaning that the values must be strictly increasing in the order that they are read (no two values are the same), or `Mono`, meaning that multiple values are allowed to be the same as long as they never decrease. For example, *3 4 5 5 6 7* would pass a `Mono` check but would fail a `Strict` check due to two of the values being the same. This rule also contains a context path that describes the relative ancestry in the input hierarchy under which the values must increase. For a simple array, this will usually be "**..**" but may go back further in lineage if needed (e.g., "**../../..**").

Schema example:

```
test{

    container{
        inside{
            IncreaseOver("../..")=Strict
        }
        badrule{
            inside{
                IncreaseOver("../..")=Neither
            }
        }
    }
    array{
        value{
            IncreaseOver("..")=Mono
        }
    }
    another_array{
        value{
            IncreaseOver("..")=Strict
        }
    }

}
```

Input example that **PASSES** validation on schema above:

```
test{

    container{
```

```
        inside=19.4
    }
    container{
        inside=24.9
    }
    container{
        inside=93.3
    }
    container{
        inside=193.3
    }
    array=[ 4.5 87.3 87.3 87.3 98.2 100.2 100.2 163.2 ]

}
```

Input example that **FAILS** validation on schema above:

```
test{

    container{
        inside=19.4
    }
    container{
        inside=24.9
    }
    container{
        inside=24.9
    }
    container{
        inside=93.3
        badrule{
        }
    }
    array=[ 4.5 87.3 87.3 87.3 48.2 100.2 100.2 63.2 ]
    array=[ 4.5 87.3 87.3 something 48.2 100.2 100.2 63.2 ]
    another_array=[ 4.5 87.3 something 87.3 48.2 100.2 100.2 63.2 ]

}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
Validation Error: Invalid Schema Rule: Bad IncreaseOver Option "Neither"
at line:9 column:39 - Expected [ Mono Strict ]

line:1 column:1 - Validation Error: test children "inside" are not strict
ly increasing at line:10 column:9

line:17 column:5 - Validation Error: array children "value" are not monot
onically increasing at line:17 column:32
```

```
line:17 column:5 - Validation Error: array children "value" are not monot
onically increasing at line:17 column:49

line:18 column:27 - Validation Error: array value "something" is wrong va
lue type for increasing

line:19 column:30 - Validation Error: another_array value "something" is
wrong value type for increasing
```

5.2.15 DecreaseOver Details and Examples

The ***Decrease Over*** rule must contain a required modifier flag that occurs as a parenthetical identifier and indicates the monotonicity. The flag must either be `Strict`, meaning that the values must be strictly decreasing in the order that they are read (no two values are the same), or `Mono`, meaning that multiple values are allowed to be the same as long as they never increase. For example, *7 6 5 5 4 3* would pass a `Mono` check but would fail a `Strict` check due to two of the values being the same. This rule also contains a context path that describes the relative ancestry in the input hierarchy under which the values must decrease. For a simple array, this will usually be "**..**" but may go back further in lineage if needed (e.g., "**../../..**").

Schema example:

```
test{

    container{
        inside{
            DecreaseOver("../..")=Strict
        }
        badrule{
            inside{
                DecreaseOver("../..")=Neither
            }
        }
    }
    array{
        value{
            DecreaseOver("..")=Mono
        }
    }
    another_array{
        value{
            DecreaseOver("..")=Strict
        }
    }

}
```

Input example that **PASSES** validation on schema above:

```
test{
```

```
    container{
        inside=193.3
    }
    container{
        inside=93.3
    }
    container{
        inside=24.9
    }
    container{
        inside=19.4
    }
    array=[ 163.2 100.2 100.2 98.2 87.3 87.3 87.3 4.5 ]

}
```

Input example that **FAILS** validation on schema above:

```
test{

    container{
        inside=93.3
    }
    container{
        inside=24.9
    }
    container{
        inside=24.9
    }
    container{
        inside=19.4
        badrule{
        }
    }
    array=[ 63.2 100.2 100.2 48.2 87.3 87.3 87.3 4.5 ]
    array=[ 163.2 100.2 100.2 something 87.3 87.3 87.3 4.5 ]
    another_array=[ 163.2 100.2 something 100.2 87.3 87.3 87.3 4.5 ]

}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
Validation Error: Invalid Schema Rule: Bad DecreaseOver Option "Neither"
at line:9 column:39 - Expected [ Mono Strict ]

line:1 column:1 - Validation Error: test children "inside" are not strict
ly decreasing at line:10 column:9

line:17 column:5 - Validation Error: array children "value" are not monot
onically decreasing at line:17 column:18
```

```
line:17 column:5 - Validation Error: array children "value" are not monot
onically decreasing at line:17 column:35

line:18 column:31 - Validation Error: array value "something" is wrong va
lue type for decreasing

line:19 column:33 - Validation Error: another_array value "something" is
wrong value type for decreasing
```

5.2.16 ChildAtMostOne Details and Examples

The ***Child At Most One*** rule contains multiple relative input lookup paths. Each of these lookup paths can optionally have an assigned lookup value. There may be more than one of these rules for any given element in the schema. Of the given list of elements, *at most one* must exist in the input in order for this rule to pass. If there is a lookup value associated with the lookup path, then that path's value in the input must be equal to that provided in the schema in order for that element to count toward existence.

Schema example:

```
test{

    ChildAtMostOne = [ one two three ]

    one{
    }
    two{
    }
    three{
    }
    four{
    }
    five{
        ChildAtMostOne=[ "../four" "../two" ]
    }
    six{
    }
    seven{
        ChildAtMostOne=[ "../six" ]
    }

}
```

Input example that **PASSES** validation on schema above:

```
test{
    one=1
    four=4
    six=6
}
test{
```

```
    four=4
}
test{
    two=3
}
test{
    five=5
}
test{
    seven=7
}
test{
    three=[ 2 3 4 ]
    four=5
    five=6
}
test{
    two=[ 2 3 4 ]
    five=6
}
test{
    five=6
    four=5
}
test{
    one=[ 2 3 4 ]
    seven=5
}
test{
    one=[ 2 3 4 ]
    six=6
    seven=5
}
test{
    one=[ 2 3 4 ]
    five=5
}
test{
    two=2
    six=[ 8 9 10 ]
    seven=[ 11 12 ]
}
```

Input example that **FAILS** validation on schema above:

```
test{
    one=1
    three=2
}
test{
```

```
    one=1
    two=[ 6 7 8 9 ]
    three=3
}
test{
    two=[ 1 2 3 4 ]
    three=5
}
test{
    four=[ 6 7 8 9 ]
    two=5
    five=4
}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
line:1 column:1 - Validation Error: test has more than one of: [ one two
three ] - at most one must occur

line:5 column:1 - Validation Error: test has more than one of: [ one two
three ] - at most one must occur

line:10 column:1 - Validation Error: test has more than one of: [ one two
three ] - at most one must occur

line:17 column:5 - Validation Error: five has more than one of: [ "../fou
r" "../two" ] - at most one must occur
```

5.2.17 ChildExactlyOne Details and Examples

The ***Child Exactly One*** rule contains multiple relative input lookup paths. Each of these lookup paths can optionally have an assigned lookup value. There may be more than one of these rules for any given element in the schema. Of the given list of elements, *exactly one* must exist in the input in order for this rule to pass. If there is a lookup value associated with the lookup path, then that path's value in the input must be equal to that provided in the schema in order for that element to count toward existence.

Schema example:

```
test{

    ChildExactlyOne = [ one two three ]

    one{
    }
    two{
    }
    three{
    }
    four{
    }
```

```
    five{
        ChildExactlyOne=[ "../four" "../two" ]
    }
    six{
    }
    seven{
        ChildExactlyOne=[ "../six" ]
    }

}
```

Input example that **PASSES** validation on schema above:

```
test{
    one=1
    four=4
    six=6
}
test{
    three=[ 2 3 4 ]
    four=5
    five=6
}
test{
    two=[ 2 3 4 ]
    five=6
}
test{
    one=[ 7 8 9 ]
    four=2
    five=6
}
test{
    two=2
    six=[ 8 9 10 ]
    seven=[ 11 12 ]
}
```

Input example that **FAILS** validation on schema above:

```
test{
    four=5
    five=6
}
test{
    one=1
    three=2
}
test{
    one=1
    two=[ 6 7 8 9 ]
```

```
    three=3
}
test{
    one=[ 6 7 8 9 ]
    five=9
}
test{
    two=[ 6 7 8 9 ]
    five=9
    four=7
}
test{
    three=[ 6 7 8 9 ]
    seven=9
}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
line:1 column:1 - Validation Error: test has zero of: [ one two three ] -
exactly one must occur

line:5 column:1 - Validation Error: test has more than one of: [ one two
three ] - exactly one must occur

line:9 column:1 - Validation Error: test has more than one of: [ one two
three ] - exactly one must occur

line:16 column:5 - Validation Error: five has zero of: [ "../four" "../tw
o" ] - exactly one must occur

line:20 column:5 - Validation Error: five has more than one of: [ "../fou
r" "../two" ] - exactly one must occur

line:25 column:5 - Validation Error: seven has zero of: [ "../six" ] - ex
actly one must occur
```

5.2.18 ChildAtLeastOne Details and Examples

The ***Child At Least One*** rule contains multiple relative input lookup paths. Each of these lookup paths can optionally have an assigned lookup value. There may be more than one of these rules for any given element in the schema. Of the given list of elements, *at least one* must exist in the input in order for this rule to pass. If there is a lookup value associated with the lookup path, then that path's value in the input must be equal to that provided in the schema in order for that element to count toward existence.

Schema example:

```
test{

    ChildAtLeastOne = [ one 'two/value' 'three/value' ]
```

```
    one{
    }
    two{
        value{
        }
    }
    three{
        value{
        }
    }
    four{
    }
    five{
        ChildAtLeastOne=[ "../four" "../two/value" ]
    }
    six{
        value{
        }
    }
    seven{
        ChildAtLeastOne=[ "../six/value" ]
    }

}
```

Input example that **PASSES** validation on schema above:

```
test{
    one=1
    four=4
    six=6
}
test{
    three=[ 2 3 4 ]
    four=5
    five=6
}
test{
    two=[ 2 3 4 ]
    four=5
    five=6
}
test{
    two=2
    five=6
}
test{
    two=2
    three=[ 5 6 7 ]
    six=[ 8 9 10 ]
```

```
    seven=[ 11 12 ]
}
```

Input example that **FAILS** validation on schema above:

```
test{
    four=5
    five=6
}
test{
    three=2
    five=5
}
test{
    one=1
    three=[ 5 6 7 ]
    seven=[ 11 12 ]
}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
line:1 column:1 - Validation Error: test has zero of: [ one 'two/value' '
three/value' ] - at least one must occur

line:7 column:5 - Validation Error: five has zero of: [ "../four" "../two
/value" ] - at least one must occur

line:12 column:5 - Validation Error: seven has zero of: [ "../six/value"
] - at least one must occur
```

5.2.19 ChildCountEqual Details and Examples

The ***Child Count Equal*** rule is usually used to ensure that arrays in the input have an equal number of value members. There may be more than one of these rules on any given element. This rule contains multiple relative input look paths and a required modifier flag that occurs as a parenthetical identifier. This modifier flag can be either `IfExists` or `EvenNone`. If the modifier flag is `IfExists`, then the pieces of input in the relative lookup paths must be equal only if they actually exist. However, if the modifier flag is `EvenNone`, then this stricter rule denotes that the relative input lookup path nodes in the input must be equal regardless of whether they exist or not.

Schema example:

```
test{

    ChildCountEqual(IfExists) = [ one/value   two/value  three/value ]
    ChildCountEqual(EvenNone) = [ four/value  five/value six/value   ]

    badflags{
        inside{
            ChildCountEqual           = [ three/value six/value ]
            ChildCountEqual(BadFlag)  = [ one/value   four/value ]
```

```
        }
    }

    one{
        value{
        }
    }
    two{
        value{
        }
    }
    three{
        value{
        }
    }
    four{
        value{
        }
    }
    five{
        value{
        }
    }
    six{
        value{
        }
    }

}
```

Input example that **PASSES** validation on schema above:

```
test{

    one=[ a b c ]
    one=d
    one=[ e f ]

    three=[ "!" "@" "#" ]
    three="$"
    three=[ "%" "^" ]

    four=[ red orange yellow ]
    four=green
    four=[ blue indigo ]

    five=[ canada poland england ]
    five=mexico
    five=[ italy france ]
```

```
    six=[ algebra chemistry history ]
    six=calculus
    six=[ physics geometry ]

}
```

Input example that **FAILS** validation on schema above:

```
test{

    one=[ a b c ]
    one=d
    one=[ e f ]

    two=[ 1 2 3 ]
    two=4
    two=[ 5 6 7 ]

    three=[ "!" "@" "#" ]
    three="$"
    three=[ "%" "^" ]

    four=[ red orange yellow ]
    four=green
    four=[ blue indigo ]

    six=[ algebra chemistry history ]
    six=calculus
    six=[ physics geometry ]

    badflags{
    }

}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
Validation Error: Invalid Schema Rule: Bad ChildCountEqual Option "" at l
ine:8 column:43 - Expected [ IfExists EvenNone ]

Validation Error: Invalid Schema Rule: Bad ChildCountEqual Option "BadFla
g" at line:9 column:43 - Expected [ IfExists EvenNone ]

line:1 column:1 - Validation Error: test does not have an equal number of
existing: [ one/value two/value three/value ]

line:1 column:1 - Validation Error: test does not have an equal number of
: [ four/value five/value six/value ]
```

5.2.20 ChildUniqueness Details and Examples

The ***Child Uniqueness*** rule is used quite often. Every value in this set must occur once and only once among all other values at all other paths. There may be more than one of these rules on any given element. This rule may use an optional `Abs` modifier flag that can occur as a parenthetical identifier. The `Abs` modifier flag indicates that the absolute value of all numbers that are added to the set checked for uniqueness are used. Then, even if one value is negative and the other is positive, but they have the same absolute value, then this validation check will fail. For example, if one ChildUniqueness relative input lookup path contains "*-5*" and another relative lookup input path contains "*5*" then this validation check will fail if the `Abs` modifier flag is used.

Schema example:

```
test{

    ChildUniqueness          = [ one/value                       ]
    ChildUniqueness          = [ one/value two/value             ]
    ChildUniqueness(Abs)     = [           two/value three/value ]
    badflags{
        inside{
            ChildUniqueness(BadFlag) = [ four/value              ]
        }
    }

    one{
        value{
        }
    }
    two{
        value{
        }
    }
    three{
        value{
        }
    }
    four{
        value{
        }
    }

}
```

Input example that **PASSES** validation on schema above:

```
test{

    one=[ 12 a b 11 c 0 -4 ]
    one=d
    one=e
    one=[ f -12 g h ]
```

```
    two=[ 1 2 3 -11 ]
    two=4
    two=5
    two=[ 6 7 8 ]

    three=[ "!" "@" "#" ]
    three="$"
    three="%"
    three=[ "^" "&" 0 "*" ]

}
```

Input example that **FAILS** validation on schema above:

```
test{

    one=[ a b c ]
    one=d
    one="%"
    one=[ 8 b h ]

    two=[ 1 b 3 0 ]
    two="%"
    two="*"
    two=[ 6 7 8 -3 ]

    three=[ 8 "@" c ]
    three="$"
    three="%"
    three=[ "^" b 0 -7 "*" ]

    badflags{
    }

}
```

HIVE validation messages that occur when validating the failing input shown above against the schema above:

```
Validation Error: Invalid Schema Rule: Bad ChildUniqueness Option "BadFla
g" at line:8 column:29 - Expected [ Abs ]

line:3 column:13 - Validation Error: one/value value "b" also exists at "
one/value" on line:6 column:13

line:3 column:13 - Validation Error: one/value value "b" also exists at "
one/value" on line:6 column:13

line:5 column:9 - Validation Error: one/value value "%" also exists at "t
wo/value" on line:9 column:9
```

```
line:6 column:11 - Validation Error: one/value value "8" also exists at "
two/value" on line:11 column:15

line:6 column:13 - Validation Error: one/value value "b" also exists at "
one/value" on line:3 column:13

line:6 column:13 - Validation Error: one/value value "b" also exists at "
one/value" on line:3 column:13

line:8 column:13 - Validation Error: two/value value "b" also exists at "
one/value" on line:3 column:13

line:8 column:13 - Validation Error: two/value value "b" also exists at "
three/value" on line:16 column:17

line:8 column:15 - Validation Error: two/value value "3" also exists at "
two/value" on line:11 column:17

line:8 column:17 - Validation Error: two/value value "0" also exists at "
three/value" on line:16 column:19

line:9 column:9 - Validation Error: two/value value "%" also exists at "o
ne/value" on line:5 column:9

line:9 column:9 - Validation Error: two/value value "%" also exists at "t
hree/value" on line:15 column:11

line:10 column:9 - Validation Error: two/value value "*" also exists at "
three/value" on line:16 column:24

line:11 column:13 - Validation Error: two/value value "7" also exists at
"three/value" on line:16 column:21

line:11 column:15 - Validation Error: two/value value "8" also exists at
"one/value" on line:6 column:11

line:11 column:15 - Validation Error: two/value value "8" also exists at
"three/value" on line:13 column:13

line:11 column:17 - Validation Error: two/value value "3" also exists at
"two/value" on line:8 column:15

line:13 column:13 - Validation Error: three/value value "8" also exists a
t "two/value" on line:11 column:15

line:15 column:11 - Validation Error: three/value value "%" also exists a
t "two/value" on line:9 column:9

line:16 column:17 - Validation Error: three/value value "b" also exists a
```

```
t "two/value" on line:8 column:13

line:16 column:19 - Validation Error: three/value value "0" also exists a
t "two/value" on line:8 column:17

line:16 column:21 - Validation Error: three/value value "7" also exists a
t "two/value" on line:11 column:13

line:16 column:24 - Validation Error: three/value value "*" also exists a
t "two/value" on line:10 column:9
```

5.3 INPUT ASSISTANCE DETAILS

Six of the previously described validation rules (`MaxOccurs`, `ChildAtMostOne`, `ChildExactlyOne`, `ValEnums`, `ExistsIn`, and `ValType`) and six new rules (`InputTmpl`, `InputName`, `InputType`, `InputVariants`, `InputDefault`, and `Description`) may also be used by input assistance applications to aid with input creation. They may be use for autocompletion assistance or input introspection.

5.3.1 MaxOccurs Assistance Details

The ***Maximum Occurrence*** rule rule may be used by input assistance application logic for filtering options as needed from the autocompletion list. An element should only be added up to MaxOccurs times via autocomplete. For example, if an element has MaxOccurs = 1, it can only be added once to the document. After is it added once, it is filtered from the autocompletion list.

5.3.2 ChildAtMostOne Assistance Details

The ***Child At Most One*** rule may be used by input assistance application logic for filtering options as needed from the autocompletion list. If at most one of multiple choices is allowed at any context, then as soon as one of those choices is added to the document, the others could be filtered from the autocompletion list. For example, if an element has ChildAtMostOne = [choice1 choice2 choice3] and choice2 is added, then choice1 and choice3 will not be available on the next autocomplete.

5.3.3 ChildExactlyOne Assistance Details

The ***Child Exactly One*** rule may be used by input assistance application logic for filtering options as needed from the autocompletion list. If exactly one of multiple choices is allowed at any context, then as soon as one of those choices is added to the document, the others could be filtered from the autocompletion list. For example, if an element has ChildExactlyOne = [choice1 choice2 choice3] and choice2 is added, then choice1 and choice3 will not be available on the next autocomplete.

5.3.4 ValEnums Assistance Details

The ***Value Enumerations*** rule may be used by input assistance application logic to provide a set of choices that are legal at a given context based on a static set of values supplied in the schema. For example, if an element has ValEnums = ["a" "b" "c" "d"], then those values could be provided as autocompletion options.

5.3.5 ExistsIn Assistance Details

The ***Exists In*** rule may be used by input assistance application logic to provide a set of choices that are legal at a given context based on values supplied elsewhere in the input. For example, if an element has ExistsIn = ["../../some/context1" "../../some/context2"] and the values 1, 2, 3, and 4 exist in the input at that relative context, then those values could be provided as autocompletion options.

5.3.6 ValType Assistance Details

The ***Value Type*** rule may be used by input assistance application logic to drop in a legitimate default value of the correct type for flag-values and flag-arrays if no `InputDefault` is provided in the schema. For example, the following defaults could be used:

- For an element with a ValType = Int rule, 1 may be inserted.
- For an element with a ValType = Real rule, 0.0 may be inserted.
- For an element with a ValType = String rule, 'insert-string-here' may be inserted.

To override this behavior, please see InputDefault Assistance Details.

5.3.7 InputTmpl Assistance Details

The ***Input Template*** rule may be used by input assistance application logic to pick which ***Template File*** to use for autocompletion. For example, if a context has InputTmpl = MyCustomTemplate then a template named `MyCustomTemplate.tmpl` may be used by the application for autocompletion.

5.3.8 InputName Assistance Details

The ***Input Name*** rule may be used by input assistance application logic to override the name of the actual node that the template provided by `InputTmpl` uses for autocompletion, if desired. For example, if the name of an element in the input hierarchy is `something_one`, then the name in the schema must be the same, but a template named `MySomething.tmpl` should use the name `something_two` instead for autocompletion, then `something_one` can be overridden via:

```
something_one{
    InputName = "something_two"
    InputTmpl = "MySomething"
}
```

5.3.9 InputType Assistance Details

The ***Input Type*** rule may be used by input assistance application logic to let the template provided by `InputTmpl` for autocompletion know what type to switch on, if desired. If a template can handle multiple situations in different ways, depending on the type it is dealing with, then the application can let the template know what the type of the current autocompletion context is with this rule. For example, if there is a template named `FlagTypes.tmpl` that can handle the types `FlagValue` and `FlagArray` differently, then the application can let the template know it is dealing with a `FlagValue` via:

```
flag_value_node{
    InputType = "FlagValue"
    InputTmpl = "FlagTypes"
}
```

Alternately, the application can let the same template know it is dealing, instead, with a `FlagArray` via:

```
flag_array_node{
    InputType = "FlagArray"
    InputTmple = "FlagType"
}
```

5.3.10 InputVariants Assistance Details

The ***Input Variants*** rule may be used by input assistance application logic to provide multiple choices of autocompletion templates for a single context. For example, if an element has `InputVariants = ['simple_version' 'middle_version' 'complex_version']` and `simple_version.tmpl`, `middle_version.tmpl`, and `complex_version.tmpl` exist in the template directory provided by application's grammar, then all three of those choices will be available at that context via autocomplete and use their associate templates.

5.3.11 InputDefault Assistance Details

The ***Input Default*** rule may be used by input assistance application logic to explicitly tell a template what value should be dropped in for flag-values and flag-arrays via `InputDefault = 'explicit_default_value'`. This should override the ValType logic described in ValType Assistance Details.

5.3.12 Description Assistance Details

The ***Input Description*** rule may be used by input assistance application logic to give a short one line description in the autocompletion dropdown list via `Description = 'autocomplete dropdown description'`. These descriptions can be very useful to novice users who are unfamiliar with all of the parameters at a given context.

6. SEQUENCE INPUT RETRIEVAL ENGINE (SIREN)

SIREN is a syntax for navigating and selecting document elements. It is heavily influenced by the XML XPath [https://www.w3schools.com/xml/xpath_syntax.asp] component within the XSLT standard.

For code examples using SIREN, see the SIREN interpreter tests in the code repository.

6.1 SELECTING NODES

The selection of nodes is performed via a path expression. Path expressions can be relative to a current node or absolute to the document.

An empty result set will be produced if no elements in the document match the given path expression.

Expression	Description
nodename	Selects all nodes with the name "*nodename*" that are children of the current node
/	Selects from the root of the document
.	Selects the current node
..	selects the parent of the current node

6.1.1 Selection Examples

Expression	Description
value	Selects all nodes with the name "*value*" that are children of the current node
/*value*	Selects all nodes with the name "*value*" that are children of the root of the document
./*value*	Selects all nodes with the name "*value*" from the current node
../*value*	Selects all nodes with the name "*value*" that are children of the parent of the current node

6.1.2 Predicates

Selection of document elements may require predicated search patterns that evaluate the position of value of the element.

Predicates can be used at all level of the path expression and are expressed as 1-base array indices, ranges, or strides, or token value equality.

Expression	Description
value[1]	Selects the first node with the name "*value*" that is a child of the current node
value[1:10]	Selects the first ten nodes with the name "*value*" that are children of the current node
value[1:10:2]	Selects every other node (stride of 2) with the name "*value*" that are children of the current node

child[*value* = 3.14]	Selects all nodes with the name "*child*" of the current node where the *child*'s *value* is equal to 3.14
child[*name* = 'fred']/*ear*	Selects all nodes with the name "*ear*" which are children of *child* of the current node, only when *child*'s name is 'fred'
child[*name* = 'fred']/*ear*[hairy='true']	Selects all nodes with the name "*ear*" which are children of *child* of the current node, only when *child*'s name is 'fred' and the ear is hairy

6.1.3 Selecting Unknown Nodes

Certain parts of the document may not be known. For this reason, wildcards are supported in the expression path.

Expression	Description
*	Selects all nodes that are children of the current node, regardless of name
*less	Selects all nodes that are children of the current node, where the node name is '*less*' or ends with '*less*'
less*	Selects all nodes that are children of the current node, where the node name is '*less*' or starts with '*less*'
l*s	Selects all nodes that are children of the current node, where the node name is '*ls*' or starts with '*l*' and ends with '*s*' with any character between

7. STANDARD OBJECT NOTATION (SON)

The standard object notation is a lightweight data entry syntax intended to facilitate application input acquisition.

The supported constructs are Objects, Arrays, and Keyed values. Additionally, Objects, Arrays, and Keyed values can be further disambiguated using identifiers.

SON can facilitate simple constructs such as property or configuration files using keyed-values.

7.1 KEYED-VALUE

The Keyed-Value is the simplest construct for representing information.

```
name = value
```

Where *name* is a string, = indicates assignment, and *value* can be a string, or number.

Alternatively, the colon ':' can be used to indicate assignment.

```
name : value
```

An example property store file that illustrates an application window attribute follows:

```
x=544 y=100
width = 1920
height = 1080
```

Lastly, if the keyed-value needs to be further disambiguated, then an identifier can be added.

```
name ( id ) = value
```

An example property store file that illustrates an application with attributes of two windows (main, settings) follows:

```
x(main)=544 y(main)=100
width(main) = 1920
height(main) = 1080

x(settings) = 520 y(settings) = 800
width (settings) = 120 height(settings) = 120
```

7.2 HIERARCHY VIA OBJECTS

Hierarchy or grouping can be added using the Object construct. Objects are useful in exhuming common context with a succinct handle.

Objects can have nested objects, keyed-values, and arrays in any order.

Objects have the following syntax.

```
object_name { ... }

object_name {
...
}

object_name
{
...
}
```

The example property store file above illustrates potential object use as follows.

```
main{
    x=544 y=100
    width = 1920
    height = 1080
}
settings{
    x = 520 y = 800
    width = 120 height = 120
}
```

Objects support the same identifier scheme as keyed-values.

7.3 ARRAYS OF DATA

SON supports 1d arrays of data that data can be scalar values, keyed-values, and nested-objects or arrays. Multi-dimensional data can be flattened to 1d, and application-specific context can be provided in associated keyed-value elements.

Arrays have the following syntax.

```
array [ ... ]

array [
 ...
]

array
[
...
]
```

The example property store file above illustrates potential array use as follows:

```
name  [ main settings ]
x     [  544      520 ]
y     [  100      800 ]
```

```
width [ 1920     1080 ]
height[  120      120 ]
```

Arrays support the same identifier scheme as keyed-values with the one exception that nested arrays cannot have identifiers.

8. DEFINITION DRIVEN INTERPRETER

The definition driven interpreter (DDI) provides a capability with very little syntax.

Specifically, DDI supports data hierarchy, arrays, and scalar values.

The pattern is as follows:

```
file := section*
section := name value* section*
```

where the section `name` must adhere to the pattern `[A-Za-z_]([A-Za-z0-9_\.])*`. A `value` can be an integer, real, or a quoted string. A file can have zero or more sections.

An example is as follows:

```
# Comments look like this
section_name1
   # keys have optional '='
   name = 1
   name = 3.14159
   name = "string value"

   array 1 2 3 4

   subsection1.1
   subsection1.2 1 2 3 3 4
       subsection1.2.1 "value"
          name = 2.71
```

The above example illustrates arbitrary hierarchy.

Note that the section indentation is recommended for clarity but is not required. All content except for comments could occur on the same line.

It is also evident that there is an ambiguity in the grammar. Specifically, how does one know whether a subsequent section is a subsection or a sibling section? Having two sections such as

```
section1
section2
```

is syntactically the same as having a subsection such as

```
section1
section1.1
```

which presents hurdles for user and program interpretation.

This is where and why the definition is important and required in driving the interpretation of these files.

The definition driven algorithm is straightforward.

1. Read a section name and perform the following
    1.1. If the section name is legal for the existing context, than capture the section name and push section context. Repeat steps.
    1.2. If the section name is not legal, pop the current context and repeat steps 1.1, 1.2, and 1.3 inquiries on new/parent context.
    1.3. If no context available, i.e., exhausted, ERROR.

The result is a parse tree where node names are section names.

9. HIERARCHIAL INPUT TEMPLATE EXPANSION ENGINE (HALITE)

The HALITE engine is a data-driven input template expansion engine intended to facilitate application input or data generation.

When it comes to text creation, there are typically two approaches: (1) write a program to generate the needed text, or (2) create templates and some program glue logic to read the templates and substitute attributes.

The HALITE engine attempts to bridge these approaches by providing standard glue logic. In this way, the templates preserve clarity of intended text, the templates facilitate reuse, development is streamlined by eliminating the developer and only requiring a template designer, and most importantly templates are interchangeable, allowing the same data to be used to create a different look, perhaps for a different application to consume.

The HALITE engine provides a single point, data-driven expansion capability that eliminates the need for application-specific glue logic.

A template and an optional hierarchical data set is all that is needed to expand templates into text that is usable by the end consumer.

The supported data constructs are provide by JSON and are Objects, Arrays, and Keyed values. For more about JSON syntax, see www.json.org.

The expression evaluation supports scalar and vector variable reference and mathematical expression evaluations.

9.1 TEMPLATE EVALUATION SUMMARY

This section describes the general approach used by the HALITE engine when evaluating a template and constructing the resulting text.

There are 3 primary components:

1. the template, consisting of constructs discussed below,
2. the optional data, described in hierarchal object notation, and
3. the evaluation stream, which is the destination of evaluating template constructs.

Available template constructs are as follows:

1. Static text: plain text to be emitted to the expanded result which contains no attributes or expression evaluation
2. Attributes: parameters or expressions to be evaluated, optionally formatted, and substituted into the expanded result
3. Optional Attributes: parameters' expressions that, when only present, are emitted to the expanded result.
4. Silent Attributes: parameters' or expressions that, when evaluated, are NOT emitted to the expanded result; useful for intermediate or cached attribute evaluations to be used later.
5. Iterative Attributes: expressions that are evaluated iteratively for a specified range(s) with optional separator and format.
6. Template imports: construct that imports a template into the existing template with optional use of data object.

7. Iterative template imports: construct that imports a template for each element of an array via 'using' an array or repeatedly via range variables.
8. Conditional blocks: support pre-processor style #if/ifdef/ifndef - #elseif/else - #endif conditional blocks which will only be emitted when the appropriate condition is true.

Each construct is evaluated and emitted into the evaluation stream, which can be redirected to a file when using the HALITE utility, or c++ std::ostream when using the wasphalite api.

9.2 ATTRIBUTES AND EXPRESSIONS

Attributes and expressions are delimited by an opening and closing delimiter. By default, these delimiters are '<' and '>' respectively. These are configurable via corresponding HaliteInterpreter class methods.

Example template attribute statements are:

1. `<attr>` - default delimiters, '<','>'
2. `{attr}` - custom '{' and '}'
3. `${attr}` - custom '${' and '}'
4. `#{attr}` - custom '#{' and '}'
5. `[:attr:]` - custom '[:' and ':]'
6. etc.

Formal attribute expression syntax appears as follows:

```
open_delim (name|expression) (':'['?''|'] format? separator? range* use? )?
close_delim
```

where

1. `open_delim` is configurable, with a default value of '<'
2. `?` indicates an optional attribute evaluation, which allows undefined variables to silently fail; `?` MUST OCCUR immediately after the attribute options delimiter ':'
3. `|` indicates silent attribute evaluation; conducts computation/variable creation without emitting the result to the evaluation stream; MUST OCCUR immediately after the attribute options delimiter ':'
4. 2 and 3 are optional and mutually exclusive
5. optional `format` is `'fmt=' format ';'`, and `format` is described in the section below.
6. optional `separator` is `'sep=' separator ';'`, and `separator` is emitted for all but the last evaluation iteration
7. zero or more `range` specifications where a range looks like `range_variable '=' start[,end[,stride]];`. Start, optional end and stride must be integers or attributes convertable to integers
8. the optional `use` statement facilitates scoped attribute access as depicted in scoped attribute sections below
9. `close_delim` is configurable, default of '>'

9.2.1 Silent Attributes

Attributes and expressions that must be evaluated but not placed into the evaluation stream can be specified using the silent expression indicator:

```
<attr:|>
```

9.2.2 Optional Attributes

Attributes may not be specified, and as such they must be considered optional by the template. Optional attributes appear as follows:

```
<attr:?>
```

Here the `:?` attribute option indicates that nothing being evaluated will be placed into the evaluation stream unless the `attr` is defined.

This is most useful when combined with formatting to tackle text where data may be optionally available, but when it is available, it requires context:

```
data record <x> <y> <z> <comment:? fmt=com="%s">
```

Here the `comment` is optional data, but when it is present it requires a 'com="`comment`"' to indicate context. The format statement provides the context 'com=' only when `comment` is present.

9.2.3 Attribute Patterns

Attribute names are defined as the regular expression `[A-Za-z_]([A-Za-z0-9\._])*`. Examples of these are:

1. `var`
2. `var_name`
3. `var.name`
4. `var1`
5. `var1.real`
6. etc.

If an attribute name contains character(s) that violate the regular expression, the variable name can be quoted. Examples of these are:

1. `'var(name)'`
2. `'my var(name)'`
3. etc.

If an attribute is an array of data, a 0-based index can be used to access the data element. Given data of

```
'array':["ted","fred",7, 3.14159 ]
```

the following attribute patterns are legal:

1. `array[0]` - "ted"
2. `array[1]` - "fred"
3. `array[2]` - 7
4. `size(array)` - 4

9.2.4 Example Attribute Pattern

An example attribute substitution appears as follows:

```
the <FoxSpeed> <FoxColor> fox jumped over the <DogColor> dog.
```

or

```
the <'fox speed'> <'fox color'> fox jumped over the <'dog color'> dog.
```

Here the `FoxSpeed` or `'fox speed'` attributes might be 'quick' or 'fast', the `FoxColor` might be 'red', and the `DogColor` or `'dog color'` might be 'brown' or 'black.'

9.2.5 Expressions

The HALITE Engine uses the WASP expression engine for expression evaluation which supports all regular math operators of multiplication '*', division '/', addition '+', subtraction '-', less than '.lt.', less than or equal '.lte.', greater than '.gt.', greater than or equal '.gte', equal '==', not equal '!=', and precedence '(',')'

- `regular trig functions - sin(x),cos(x),tan(x),asin(x),acos(x),atan(x),atan2(y,x),sec(x),cosec(x),cot(x),sinh(x),cosh(x),tanh(x),asinh(x),acosh(x),atanh(x)`
- `logarithmic functions - ln(x),log(x),lg(x)`
- `exp(x)`
- `pow(x)`
- `round(x),round(x, p)`
- `floor(x),ceil(x)`
- `if(cond, trueval, falseval)`
- `abs(x)`
- `modulo - mod(x,y),`
- `min(x,y) max(x,y)`
- `sqrt(x)`
- `defined('x') - indicates if the variable named x is defined`

9.2.6 Example Expression Patterns

An example attribute substitution appears as follows:

```
the quick red fox jumped over the brown dog going <miles/hour>mph fast.
```

or

```
the quick red fox jumped over the brown dog going <velocity_mph*1.60934>kph f
ast.
```

In addition to integer and double precision math operations, string concatenation is also available, such as

```
<"My result is "+numeric_result>
```

Here the `numeric_result` is concatenated to the string `My result is`, thus producing a final result that is string typed.

9.3 FORMATTING

Attribute and expressions can be formatted prior to insertion into the evaluation stream. This is influenced by the C printf and Java.Format capability.

Specifically, the following constructs are provided:

```
%[flags][width][.precision]specifier
```

9.3.1.1 Format Specifiers

The available specifiers are shown below:

Specifier	Description	Example
f	Decimal floating point	3.14159
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
s	String of characters	sample
d	Signed decimal value	300

The format declarator percent % can be escaped with a double percent specified, %%

9.3.1.2 Format Flags

The available flags are presented below

flag	Description	Example
-	The result will be left justified	'left '
+	The result will always include a sign	'+result'
' '	The result will include a leading space for positive values	' result'
0	The result will be zero padded	'0result'
(	The result will enclose negative numbers in parentheses	'(negative result)'

9.3.1.3 Format Width

The width is the minimum number of characters to be written to the output. Most frequently used for padding.

9.3.1.4 Format Precision

For general argument types the precision is the maximum number of characters to be written to the output. For floating-point types (specifier = 'e','f'), the precision is the number of digits after the decimal point.

9.3.1.5 Format Examples

String examples are shown below:

Format	Result	Description
'<"str":fmt=%s>'	'str'	Print the raw string "str" as a string
'<"str":fmt=%4s>'	' str'	Print the raw string "str" with a width of 4 as a string
'<"str":fmt=%05s>'	'00str'	Print the raw string "str" with a width of 5 as a string padded with zeros
'<"-30":fmt=%05s>'	'00-30'	Print the raw string "-30" with a width of 5 as a string padded with zeros
'<"-30":fmt=%%05s="%05s">'	'%05s="00-30"'	Print the raw string "-30" with a format prefix of '%05s', a width of 5 as a string padded with zeros
'<"str":fmt=%-10s>'	'str '	Print the raw string "str" left justified with a minimum width of 10

Integer examples are listed below:

Format	Result	Description
'<3:fmt=%d>'	'3'	Print the integer 3 as an integer
'<30:fmt=%4d>'	' 30'	Print the integer 30 with a width of 4 as an integer
'<-30:fmt=%-5d>'	'-30 '	Print the integer -30 left justified with a width of 5 as an integer
'<-30:fmt=%05d>'	'-0030'	Print the integer -30 with a width of 5 as an integer padded with zeros
'<30:fmt=% d>'	' 30'	Print the integer 30 with a leading space due to positive value
'<x=-30:fmt=% d>'	'-30'	Print the variable x (-30) and if x > 0 include a leading space
'<30:fmt=%+d>'	'+30'	Print the integer 30 with its sign
'<x=-30:fmt=%(d>'	'(30)'	Print the variable x (-30) with parenthesis if x < 0
'<3.14159:fmt=%d>'	'3'	Print the floating point value as an integer

Float-Point examples are as follows:

Format	Result	Description
'<3.14159265:fmt=%f>'	'3.141593'	Print the floating-point value 3.14159265 as a floating point value with default precision of 6
'<3.14159265:fmt=%7f>'	'3.141593'	
'<3.14159265:fmt=%8f>'	'3.141593'	
'<3.14159265:fmt=%9f>'	' 3.141593'	
'<3.14159265:fmt=%10f>'	' 3.141593'	
'<3.14159265:fmt=%.0f>'	'3'	
'<3.14159265:fmt=%.1f>'	'3.1'	
'<3.14159265:fmt=%.2f>'	'3.14'	
'<3.14159265:fmt=%.3f>'	'3.142'	
'<3.14159265:fmt=%1.0f>'	'3'	
'<3.14159265:fmt=%4.1f>'	' 3.1'	
'<3.14159265:fmt=%4.8f>'	'3.14159265'	
'<3.14159265:fmt=%8.2f>'	' 3.14'	
'<3.14159265:fmt=%10.3f>'	' 3.142'	
'<3.14159265:fmt=%01.0f>'	'3'	
'<3.14159265:fmt=%04.1f>'	'03.1'	
'<3.14159265:fmt=%04.8f>'	'3.14159265'	
'<3.14159265:fmt=%08.2f>'	'00003.14'	
'<3.14159265:fmt=%010.3f>'	'000003.142'	
'<3.14159265:fmt=%g>'	'3.14159'	
'<3.14159265:fmt=%.0g>'	'3'	
'<3.14159265:fmt=%.1g>'	'3'	
'<3.14159265:fmt=%.2g>'	'3.1'	
'<3.14159265:fmt=%.3g>'	'3.14'	
'<3.14159265:fmt=%1.0g>'	'3'	
'<3.14159265:fmt=%4.1g>'	' 3'	
'<3.14159265:fmt=%4.8g>'	'3.1415927'	
'<3.14159265:fmt=%8.2g>'	' 3.1'	
'<3.14159265:fmt=%10.3g>'	' 3.14'	
'<3.14159265:fmt=%01.0g>'	'3'	
'<3.14159265:fmt=%04.1g>'	'0003'	
'<3.14159265:fmt=%04.8g>'	'3.1415927'	

```
'<3.14159265:fmt=%08.2g>'    '000003.1'
'<3.14159265:fmt=%010.3g>'   '0000003.14'
'<1e-4:fmt=%.4g>'            '0.0001000'
'<1e-4:fmt=%10.4g>'          ' 0.0001000'
'<1e-4:fmt=%.5g>'            '0.00010000'
'<1e-5:fmt=%.5g>'            '1.0000e-05'
'<1e-6:fmt=%.6g>'            '1.00000e-06'
'<1e-2:fmt=%.7g>'            '0.01000000'
'<1e2:fmt=%10.7g>'           '  100.0000'
'<3.14159265:fmt=%e>'        '3.141593e+00'
'<3.14159265:fmt=%.0e>'      '3e+00'
'<3.14159265:fmt=%.1e>'      '3.1e+00'
'<3.14159265:fmt=%.2e>'      '3.14e+00'
'<3.14159265:fmt=%.3e>'      '3.142e+00'
'<3.14159265:fmt=%1.0e>'     '3e+00'
'<3.14159265:fmt=%4.1e>'     '3.1e+00'
'<3.14159265:fmt=%4.8e>'     '3.14159265e+00'
'<3.14159265:fmt=%8.2e>'     '3.14e+00'
'<3.14159265:fmt=%10.3e>'    ' 3.142e+00'
'<3.14159265:fmt=%01.0e>'    '3e+00'
'<3.14159265:fmt=%04.1e>'    '3.1e+00'
'<3.14159265:fmt=%04.8e>'    '3.14159265e+00'
'<3.14159265:fmt=%08.2e>'    '3.14e+00'
'<3.14159265:fmt=%010.3e>'   '03.142e+00'
'<1e-4:fmt=%.4e>'            '1.0000e-04'
'<1e-4:fmt=%10.4e>'          '1.0000e-04'
'<1e-4:fmt=%.5e>'            '1.00000e-04'
'<1e-5:fmt=%.5e>'            '1.00000e-05'
'<1e-6:fmt=%.6e>'            '1.000000e-06'
'<1e-2:fmt=%.7e>'            '1.0000000e-02'
'<1e-2:fmt=%10.7e>'          '1.0000000e-02'
'<1e2:fmt=%10.7e>'           '1.0000000e+02'
```

9.3.1.6

9.3.1.7 Format Error Examples

Some flag specifiers are not legal with given specifiers. The following are examples of common scenarios:

Format	Error
'<"some string":fmt=%(5s>'	`malformed format string: flag '(' does not match the conversion 's'`
'<"some string":fmt=%+5s>'	`malformed format string: flag '+' does not match the conversion 's'`
'<"some string":fmt=% s>'	`malformed format string: flag ' ' does not match the conversion 's'`

9.4 SCOPED ATTRIBUTE

Because the data is hierarchical, access to lower levels of the data hierarchy can be required. Common single-level access is facilitated by scoping an attribute access via a 'use' statement.

9.4.1 Object Scoped Attribute

Given hierarchical data of

```
{
   'fox' : { 'color' : 'red', 'speed' : 'quick' }
   ,'dog color' : 'brown'
}
```

the following template uses scoped attributes of `fox` `color` and `speed` to emit the desired result:

```
the <speed:use=fox> <color:use=fox> fox jumped over the <'dog color'> dog.
```

or

```
<"the "+speed+" "+color:use=fox> fox jumped over the <'dog color'> dog.
```

9.4.2 Array Scoped Attribute

In addition to object-scoped use statements, an array can be used. When an array is used, an iteration is implied over each element of the array.

Given the array data of

```
{
    'parts' :
    [
        { "what" : "quick red fox", "action" : "jumped over", "dog state" : "
brown" }
        ,{ "what" : "honey badger", "action" : "didn't care about", "dog stat
e" : "big angry" }
        ,{ "what" : "weasel", "action" : "slunk by", "dog state" : "sleeping"
}
```

```
    ]
}
```

the following template uses the scoped attributes, as follows:

```
<"The "+what+" "+action+" the "+'dog state'+" dog.":use=parts>
```

The result is a whitespace-separated evaluation of the template using each element in the array:

```
The quick red fox jumped over the brown dog. The honey badger didn't care abo
ut the big angry dog. The weasel slunk by the sleeping dog.
```

A file import using an object or array facilitates more complex hierarchical data access.

9.5 FILE IMPORTS

The HALITE engine supports file import, where files consist of all template constructs described in this section. File imports can be parameterized and both implicitly and explicitly iterative.
The simplest file format appears as shown below:

```
#import relative/or/absolute/file/path.tmpl
```

Here the '#import' must occur at the start of the line. The path to the file can be relative to the current template, working directory, or an absolute path.

The capability for the file being relative to the working directory allows subtemplates to be overridden.

The path can also be templated on any available attribute. The subtemplate has immediate access to all attributes at the current data level.

9.5.1 Example Data

The given data are presented below:

```
{
    "x" : "blurg"
    ,"y" : "blarg"
    ,obj :
    {
        "a" : "blurgit", "e" : "blarg" ,
        "sarg" :
        {
            "bravo" : 2
            , "delta" : 4
            , "charlie" : 3
        }
        ...
    }
    ,array :
    [
        {...}
        ,...
```

```
    ]
}
```

The root template (entry for evaluation) has access to `x`, and `y`, but in order to access data members of `obj`, either a scoped attribute evaluation or a parameterized template import must be used.

9.5.2 Parameterized File Import

Parameterized file imports facilitate access to data hierarchy and repetition of templates.

9.5.2.1 File Import Using an Object

The example data contains the `obj` data layer, which contains nested object `sarg` and other imaginary data `....` To access all of the data, a subtemplate can be imported 'using' `obj` as follows:

```
#import some/file.tmpl using obj
```

The template `some/file.tmpl` can now access all attributes within `obj`. Additionally, all attributes in higher levels (`x`,`y`,`array`,`...`) are still accessible:

`some/file.tmpl`:

```
This is a nested template with access to obj's context
variable a=<a>, e=<e>

Variables still accessible from parent data are x:<x>, and y:<y>, etc.
```

9.5.2.2 Iterative File Import Using an Array or Ranges

The import of files can be repeated using 2 constructs: (1) implicit iteration via use of an array, or (2) explicit iteration using repeated ranges.

The implicit iteration via use of an array is syntactically identical to import using an object:

```
#import path/to/file.tmpl using my_array
```

The explict iteration via ranges is syntactically different to disambiguate and clearly indicates intent:

```
#repeat path/to/file.tmpl using (var=start[,end[,stride]];)+
```

Note that the statement starts with `#repeat` and not the regular `#import`. The range (`var`) can be specified as semicolon ';' delimited to produce embedded loops.

```
#repeat path/to/file.tmpl using i=1,5; j=2,6,2;
```

The above will loop `j`=2,4,6 for `i`=1 through 5. The variables `i` and `j` are available in the imported template.

9.6 CONDITIONAL BLOCKS

Conditional blocks facilitate alternative paths through templates. Conditional blocks can be activated with a defined or undefined variable or an expression that evaluates to true or false.

`#if`, `#ifdef` and `#ifndef` indicate the start of a conditional block. Subsequently, an additional condition can be indicated by `#elseif` or `#else` and then finally terminated by a `#endif`.

```
#if condition
block
#endif
```

The above block will be emitted if the `condition` evaluates to true. The `condition` could be a variable name referencing a value that is true, or it could be an attribute expression.

```
#if ted_present
hello ted
#endif
```

or

```
#if < i .gt. 0 >
some logic pertaining to i > 0
#endif
```

The use of `#elseif` and `#else` allow for alternative logic if the initial or prior condition evaluates to false.

```
#if < ted_present && bill_present >
Bill and Ted's excellent adventure!
#elseif ted_present
Where is Bill?
#elseif bill_present
Where is Ted?
#else
Where are Bill and Ted?
#endif
```

10. COMMAND LINE UTILITIES

The Workbench Analysis Sequence Processor package provides a set of command line utilities to aid in sequence processing and processor development.

The sequence processor construct parses trees from which two primary functions are typically desired:

1. the listing of the parse tree
2. the selection of input given a select statement.

10.1 FILE LISTING UTILITIES

Available Interpreters have corresponding *list utilities.

SON, GetPot, DDI, and JSON have corresponding sonlist, getpotlist, ddilist, and jsonlist.

These utilities produce an ordered directory-style listing of each parsed input component.

For the given example SON data file `example.son`:

```
object(identifier){
   key = value
   child ( name ) {
      x = 1
   }
}
array [ 1 2 3 ]
```

an invocation of the `sonlist` utility,

```
sonlist example.son
```

produces a directory-style list of each component in the file:

```
/
/object
/object/decl (object)
/object/( (()
/object/id (identifier)
/object/) ())
/object/{ ({)
/object/key
/object/key/decl (key)
/object/key/= (=)
/object/key/value (value)
/object/child
/object/child/decl (child)
/object/child/( (()
/object/child/id (name)
/object/child/) ())
/object/child/{ ({)
```

```
/object/child/x
/object/child/x/decl (x)
/object/child/x/= (=)
/object/child/x/value (1)
/object/child/} (})
/object/} (})
/array
/array/decl (array)
/array/[ ([)
/array/value (1)
/array/value (2)
/array/value (3)
/array/] (])
```

These lists describe the interpreted hierarchy and value.

Note that the Definition-Driven Interpreter (DDI) is different from the others, as it requires a schema (definition) in order to parse.

```
ddilist /path/to/schema.sch /path/to/input.inp
```

10.2 FILE COMPONENT SELECTION UTILITIES

The ability to select specific parts of the input can be useful in schema creation. All supported interpreters have corresponding select utilities.

Using the example.son file earlier, a select statement of `sonselect example.son /array/value[1:3]` produces the following output:

```
Selecting /array/value[1:3]
---- 3 nodes selected with statement '/array/value[1:3]' ----
1) /array/value
1
2) /array/value
2
3) /array/value
3
```

Alternatively, `sonselect example.son /object/child/x` produces the following output:

```
Selecting /object/child/x
---- 1 nodes selected with statement '/object/child/x' ----
1) /object/child/x
x = 1
```

Subsequent selections select from prior selection sets, so `sonselect example.son /object/child/x ../key/value` produces:

```
Selecting /object/child/x
---- 1 nodes selected with statement '/object/child/x' ----
1) /object/child/x
x = 1
```

```
Selecting ../../../array
---- 1 nodes selected with statement '../../../array' ----
1) /array
array [ 1 2 3 ]
```

The first node selected is the `/object/child/x` node, which has the text `x = 1`. Subsequently, the relative path `../../../array` is used from `/object/child/x` to select three levels up `../../../`, and subsequently the `array` node. Notice that the exact user input is reproduced.

10.3 XML UTILITIES

The XML standard is readily accessible in most programming languages where SON, GetPot, DDI, etc. are not. As such, the *xml utilities provide a bridge for prototyping or coupling with higher level scripts, etc.

```
sonxml example.son
```

produces

```
<document>
  <object>
    <decl loc="1.1" dec="true">object</decl>
    <LP loc="1.7" dec="true">(</LP>
    <id loc="1.8" dec="true">identifier</id>
    <RP loc="1.18" dec="true">)</RP>
    <LBC loc="1.19" dec="true">{</LBC>
    <key>
      <decl loc="2.4" dec="true">key</decl>
      <ASSIGN loc="2.8" dec="true">=</ASSIGN>
      <value loc="2.10">value</value>
    </key>
    <child>
      <decl loc="3.4" dec="true">child</decl>
      <LP loc="3.10" dec="true">(</LP>
      <id loc="3.12" dec="true">name</id>
      <RP loc="3.17" dec="true">)</RP>
      <LBC loc="3.19" dec="true">{</LBC>
      <x>
        <decl loc="4.7" dec="true">x</decl>
        <ASSIGN loc="4.9" dec="true">=</ASSIGN>
        <value loc="4.11">1</value>
      </x>
      <RBC loc="5.4" dec="true">}</RBC>
    </child>
    <RBC loc="6.1" dec="true">}</RBC>
  </object>
  <array>
    <decl loc="7.1" dec="true">array</decl>
    <LBK loc="7.7" dec="true">[</LBK>
    <value loc="7.9">1</value>
    <value loc="7.11">2</value>
```

```
    <value loc="7.13">3</value>
    <RBK loc="7.15" dec="true">]</RBK>
  </array>
</document>
```

Any xml element with the attribute `dec="true"` indicates a 'decorative' input component, required syntax and could be ignored by most higher-level interpreters.

The attribute `loc="line.column"` indicates the input components location in the file.

The element's name indicates the name of the input component. The leaf element's data are the data of interest stored in the parse tree.

10.4 FILE VALIDATION UTILITIES

The SON, DDI, and GetPot interpreters have Hierarchical Input Validation Engine (HIVE) adapters allowing them to be validated. As such, there are sonvalid, ddivalid, and getpotvalid utilities.

Invocation of the validation utilities requires a schema, and an input:

```
sonvalid /path/to/schema.sch /path/to/input.inp
```

The schema's contents are beyond the scope of this document. The product of *valid will be a return code of 0 only if no validation errors occur in the input. If an error occurs, a non-zero return code is produced, and validation errors emitted. See the HIVE documentation of types of validation errors.

10.5 THE HIERARCHAL INPUT TEMPLATE EXPANSION (HALITE) ENGINE

The HALITE engine has the corresponding `halite` command line utility.

HALITE provides a data-driven template expansion capability and has a sizable feature set for templating text data for input or other needs.

The `halite` command line utility can be invoked with a template and optional JSON-formatted data parameter set:

```
halite /path/to/template.tmpl
```

or with JSON parameter set:

```
halite /path/to/template.tmpl /path/to/data.json
```

The expanded template emitted on stdout, and errors/log information is emitted on stderr.

10.6 SCHEMA SKELETON CREATION UTILITY

The ability to take multiple input files known to be valid and to create a schema skeleton from these can be a very useful starting point for schema creation. Currently, a utility exists that allows this to be done with a series of SON input files. Therefore, if a user has multiple SON formatted input files that are known to be valid, and the objective is to begin creating a schema for these files, the first step is to run

```
sonschemaskel path/to/valid/input1.son path/to/valid/input2.son ...
```

and the resulting output will be a schema skeleton that can be used with the Hierarchical Input Validation Engine to validate the inputs. These schema skeletons have actual rule stubs for each input node commented out. These rules can be reviewed in more detail in the HIVE section and should be modified for each piece of input.