SLAT Language

Michael Gauland*

2019-03-13 12:12

This document describes the command language used by the `SLAT` interpreter. It is very much a work in progress; please let me know of any conflicts, omissions, ambiguities, or other problems.

1 Language Description

This section describes the general structure of the language, and includes examples of both commands and typical output. A formal definition of the language will follow.

Commands consist of a keyword, possibly followed by a combination of:

- an identifier, if something is being created
- additional keywords, further specifying the command
- identifiers, indicating the variables to operate on
- additional parameters, such as the numbers over which to evaluate a function
- optional flags and parameters controlling how the command behaves
- the end of a command is indicated by a semi-colon (';') (this does not apply to comments)

1.1 Comments

The hash symbol ('#') starts a **comment**, which goes to the end of the line:

```
# This is a comment; it will be ignored.

title "Example␣1"; # This comment follows a command

# Comments can also appear inside a command:
fragfn frag1 [0.0062, 0.4],    # Damage State 1
             [0.0230, 0.4],    # Damage State 2
             [0.0440, 0.4],    # Damage State 3
```

*michael.gauland@canterbury.ac.nz

```
                [0.0564, 0.4];    # Damage State 4
```

1.2 Title

The `title` command creates a title for the anaylsis. I'm not yet sure where this will be used, though I expect it may be included in output files or on plots.

The title must be in quotes (single or double):

```
title "Example␣1"
title 'Example␣2;␣Single␣Quote␣are␣also␣acceptable'
title 'Example␣3;␣opposite-type␣"quotes"␣can␣be␣nested'
title "Example␣4--'in␣either␣order'"
```

Strings can also use escape sequences to include quotes or backslashes:

```
title 'Single-quoted␣strings␣can␣escape␣\'␣and␣\\.';
title "Double-quoted␣strings␣can␣escape␣\"␣and␣\\.";
```

If more than one title command is supplied, the strings will be treated as separate lines of a single title.

The title can be explicitly included in output with the `print title` command.

1.3 Deterministic Functions

Deterministic functions are defined with the `detfn` command:

```
detfn <detfn_id> <function type> <parameters>;
```

The command must specify the function type, the name under which to store the function, and the parameters:

```
# Define the hyperbolic function that describes the
# IM-Rate relationship:
#
#                              v_asy  im_asy   alpha
detfn IM_FUNC_ID hyperbolic 1221,    29.8,   62.2;
```

The `hyperbolic` deterministic function type expects three numbers: `v_asy`, `im_asy`, and `alpha`, in that order.

The equation for the non-linear hyperbolic law is:

$$y = v_{asy} \cdot e^{\frac{\alpha}{\ln \frac{x}{im_{asy}}}} \tag{1}$$

In addition to `hyperbolic`, `detfn` also accepts `powerlaw` as a function type:

```
# POWERLAW functions ca be used to describe the MU and
# SIGMA of the EDP-IM relationship:
detfn powerlaw    mu_func  0.1, 1.5;
detfn powerlaw sigma_func  0.5, 0.0;
```

The two arguments a and b are are used like this:

$$y = a \cdot x^b \tag{2}$$

1.4 Probabilistic Functions

Probabilistic functions are defined with the `probfn` command. `lognormal` may be the only variant needed, but other distributions could easily be added. The definition identifies the functions which define the distribution:

```
probfn <probfn_id> lognormal <mu_func_id> <sd_func_id>
                   <placement_type>
                   <spread_type>;
```

where `mu_func_id` and `sd_func_id` refer to deterministic functions defined with the `detfn` command. *<placement_type>* is optional; if present it consists of the flag `--mu`, followed by one of:

- `median_x` : the *<mu_func_id>* function describes the median of the distribution.
- `mean_x` : the *<mu_func_id>* function describes the mean of the distribution.
- `mu_ln_x` : the *<mu_func_id>* function describes the mean of the log of distribution.

By default, *<mu_func_id>* describes the mean of the log of the distribution.

Similarly, `spread_type` is also optional. If present it consists of the flag `--sd`, followed by one of:

- `sd_x` : the *<sd_func_id>* function describes the standard deviation of the distribution.
- `sd_ln_x` : the *<sd_func_id>* function describes the standard deviation of log of the distribution.

By default, *<sd_func_id>* describes the standard deviations of the log of the distribution.

This command uses the deterministic functions `mu_func` and `sd_func` to define the probabilistic function `edp_fn`, with both functions referring to the log of the distribution:

```
probfn lognormal edp_fn mu_func sd_func;
```

This command uses the same functions, but mu_func describes the median of the distribution.

```
probfn edp_fn lognormal mu_func sd_func --mu median_x;
```

1.5 Relationships

I'm using the term 'relationship' to describe how the various functions are connected. Would another term be clearer?

An IM relationship simply refer s to a deterministic function:

```
im  <im_id>  <detfn_id>;
```

For example:

```
im im1 im_func;
```

An EDP relationship refers to an IM relationship, and the probabilistic function describing the engineering response:

```
edp <edp_id> <im_id> <probfn_id>;
```

For example:

```
edp edp1 im1 edp_fn;
```

1.6 Fragility Functions

Fragility functions can be specified as a set of parameters, describing the log-normal distribution of the onset of each state, such as:

```
fragfn FRAGID [0.0062, 0.4], # Damage state #1
              [0.0230, 0.4], # Damage state #2
              [0.0440, 0.4], # Damage state #3
              [0.0564, 0.4]; # Damage state #4
```

Each pair numbers represents the μ and σ values of the onset for a damage state. By default, these describe the mean and standard deviation of the log of the distribution, but can be changed as can be done for probabilistic functions:

```
fragfn FRAGID [0.0062, 0.4], [0.0230, 0.4],
              [0.0440, 0.4], [0.0564, 0.4]
       --sd sd_x  --mu mean_x;
```

Fragility functions can also be read from a database of standard functions:

```
fragfn FRAGID --stdfunc FRAG_KEY;
```

FRAG_KEY identifies the fragility function in the database. The format of keys is still to be decided.

It is also possible to use fragility functions from custom databases:

```
fragfn FRAGID --db FILENAME.EXT --stdfunc FRAG_KEY;
```

I haven't yet given any thought to the structure of the database, or how to identify functions within it.

1.7 Loss Functions

I expect to eventually support a variety of loss function types, but for now only simple is supported. This assumes a fixed per-unit cost, following a lognormal distribution described by μ and σ. As with other lognormal distributions, μ and σ describe the mean and standard deviation of the log of the distribution, but can be changed as can be done for probabilistic functions

```
lossfn LOSS1 simple [0.03, 0.4], # Damage state #1
                    [0.08, 0.4], # Damage state #2
                    [0.25, 0.4], # Damage state #3
                    [1.00, 0.4]; # Damage state #4
```

```
lossfn LOSS1 simple [0.03,0.4], [0.08, 0.4],
                    [0.25, 0.4], [1.00, 0.4]
             --mu mean_x --sd sd_ln_x;
```

1.8 Component Groups

A component group specifies a number of components sharing the same EDP relationship, fragility function, and loss function:

```
comgrp <comgrp_id> <edpid> <fragid> <lossid> <count>
```

For example:

```
comgrp cgroup1 edp1 frag1 loss1 10;
```

1.9 Print Command

The print command is used to print definitions to the terminal or a file:

```
print <definition type> <identifier[s]> <options>;
```

where <definition type> is one of:

- detfn
- probfn
- im
- edp
- fragfn
- lossfn
- comgrp

We must specify the type of definition to be printed, since identifiers can be re-used across types of definitions.

print can also be used to print a message:

```
print message <string> <options>;
```

...or the title:

```
print title <options>:;
```

<target> can be:

- empty (i.e., it is optional), which sends the output to stdout.
- a file name, indicating where to send the output. This can optionally be followed by:
  - The flag new, which is the default. This means the output file will be silently overwritten if it already exists.
  - The flag append, which indicates the data should be appended to the end of the file, if it already exists.

We can make additional options available as needed.

Some examples:

```
print message;
print message "This␣is␣a␣message␣string." MESSAGE.TXT;
print message 'This␣is␣a␣message.' MESSAGE.TXT --new;
print message 'This␣is␣a␣message.' MESSAGE.TXT --append;
print detfn IM_FUNC INPUT_DEFS.TXT --new;
print probfn EDP_FN INPUT_DEFS.TXT --append;
print im IM1 INPUT_DEFS.TXT --append;
print edp EDP1 INPUT_DEFS.TXT --append;
print fragfn FRAG1 INPUT_DEFS.TXT --append;
print lossfn LOSS1 INPUT_DEFS.TXT --append;
print compgroup PG1 INPUT_DEFS.TXT --append;
```

print is executed right away–it does not wait for the analyze command. This can help with troubleshooting, if the system crashes during an analysis.

1.10 Integration

The integration command sets up integration parameters

```
integration <algorithm> <accuracy> <iterations>;
```

For example:

```
integration maq 1E-6 1024;
```

1.11 Recorders

The recorder command specifies that a function or relationship should be recorded over specified values, to a file or the terminal:

```
recorder <type> <id> <at-values> <columns> <options>;
```

<type> and <id> are required. <type> must be one of:

- detfn
- probfn
- imrate
- edpim
- dsedp
- dsim
- dsrate
- lossds
- lossedp
- lossim

<id> identifies the function for the recorder to use (the function type depends on the recorder <type>; more on that below).

<at-values> is requires for all but recorder dsrate, to identify the values at which the function is to be recorded. It can be:

- Three numbers, separated by colons, indicating the starting value, ending value, and step size:

  ```
  recorder detfn im_func 0.0:1.0:0.1;
  ```

  will record detfn im_func at values from 0 to 1, in steps of 0.1.

- One or more numbers, separated by commas, enumerating the values to use:

```
recorder detfn im_func 0.0, 0.1, 0.2;
```

will record `detfn` `im_func` at 0.0, 0.1, and 0.2.

- A Python expression that evaluates to a set of values:

```
recorder detfn im_func
         $(numpy.arange(0.0, 0.011, 0.001));
```

will record `detfn` `im_func` for values from 0.00 to 0.01, in steps of 0.001. Note that the `numpy.arange` function excludes the ending limit.

- A variable that evaluates to a set of values:

```
set values $(numpy.arange(0.0, 1.01, 0.01));
recorder imrate IM_1 $values;
```

will record `detfn` `im_func`, at the values stored in the variable `values`.

The `cols:`*<values>* clause is optional, and is only relevent for types which represent probability distributions (`probfn`, `edp`, `fragfn`, and `lossfn`).

The `at:`*<values>* clause specifies the values at which to provide output. This can be specified in several ways:

- As a list of numbers:

```
recorder detfn im_func at:[0.0, 0.1, 0.2, 0.3]
```

- As three numbers, indicating the start value, increment, and end value:

```
recorder detfn im_func AT:<min>:<incr>:<max>
```

For example, this:

```
recorder detfn im_func AT:0.0:0.001:0.010
```

will report data for x values from 0.0 to 0.01 (inclusive), in steps of 0.001.

- As a snippet of python code that evalues to a list of values:

```
recorder detfn im_func
         at:$(numpy.arange(0.0, 0.011, 0.001))
```

and

```
recorder detfn im_func
         at:$(numpy.arange(start=0.0,
                           step=0.001, stop=0.011))
```

are both equivalent to the previous example.[1]

The use of Python code in commands will be discussed below.

1.11.1 detfn

```
recorder detfn im_func at:[0.0, 0.1, 0.2]
```

X	im_func(x)
0.0	1.000
0.1	0.022
0.2	0.004

The calculations required depend on the function type.

1.11.2 probfn

By default, a probability function reports on the parameters which were used to define the distribution (that is, median_X, mean_X, or mu_lnX and sd_X or sigma_lnX:

```
recorder probfn edp_fn at:0.0:0.1:0.5
```

X	mu_ln(edp_fn(X))	sigma_ln(edp_fn((X))
0.0	0.000	0.500
0.1	0.003	0.500
0.2	0.008	0.500
0.3	0.016	0.500
0.4	0.025	0.500
0.5	0.035	0.500

Another option is to provide a cols parameter, providing one or more probabilities:

```
recorder probfn edp_fn COLS:[0.16, 0.50, 0.84] AT:0.0:0.1:0.5
```

IM	0.16	0.50	0.84
0.0	0.000	0.000	0.000
0.1	0.002	0.003	0.005
0.2	0.005	0.009	0.015
0.3	0.010	0.016	0.027
0.4	0.015	0.025	0.042
0.5	0.022	0.035	0.058

The cols parameter can also include the symbols used to define probabilistic functions (see section 1.4).

[1] Note that the end limit is *not* included in the results of the `arange` function.

```
recorder probfn edp_fn cols:[mu_lnX, 0.16, sigma_lnX]
                        at:0.0:0.1:0.5
```

X	mean_ln(edp_fn(X))	0.16	sigma_ln(edp_fn((X))
0.0	0.000	0.000	0.500
0.1	0.003	0.002	0.500
0.2	0.008	0.005	0.500
0.3	0.016	0.010	0.500
0.4	0.025	0.015	0.500
0.5	0.035	0.022	0.500

Calculating the distribution parameters is simply a matter of evaluating the deterministic functions used to define them (an possibly translating them, if the parameters requested do not match those stored internally).

Calculating the value at a given probability from the distribution parameters is done by the GNU Scientific Library (GSL) function `gsl_cdf_lognormal_Pinv()`. I haven't looked at the code for this, but I don't expect to be able to improve on it.

1.11.3 imrate

An `imrate`-type `recorder` evaluates the exceedence function for given values of the intensity measure:

```
recorder imrate im1 at:[0.0, 0.1, 0.2]
```

IM	Exceedence(IM)
0.0	1.000
0.1	0.022
0.2	0.004

These calculations simply require evaluating the deterministic function used to define the intensity measure.

1.11.4 edpim

An `edpim`-type `recorder` reports on the distribution of the engineering demand parameter for given values of the intensity measure. Since this is a probabilistic function, it accepts a `cols` parameter as described above.

```
recorder edpim edp1 cols:[mean_X, sd_X, 0.16]
                      at:0.0:0.1:0.5
```

Calculating the distribution parameters is simply a matter of evaluating the probabilistic function used to define the engineering demand parameter.

IM	mean_X(EDP(IM))	sd_X(EDP(IM))	0.16
0.0	0.000	0.500	0.000
0.1	0.003	0.500	0.002
0.2	0.008	0.500	0.005
0.3	0.016	0.500	0.010
0.4	0.025	0.500	0.015
0.5	0.035	0.500	0.022

Calculating the percentage points is easily done from the distribution parameters, using standard statistics functions (e.g., `qlnorm()` in `R`).

1.11.5 edprate

An `edprate`-type `recorder` reports on the rate of exceedence for given values of the engineering demand parameter.

```
recorder edprate edp1 at:0.001:0.001:0.005
```

EDP	Exceedence
0.001	0.088
0.002	0.043
0.003	0.027
0.004	0.019
0.005	0.014

For a given IM value im, the probability that the EDP will exceed a given EDP value e (written as "$p(EDP > e|IM = im)$") is easily calculated from the distribution parameters for EDP at im, using standards statistics functions (such as `plnorm()` in `R`)

The probability of exceeding a given EDP value e is

$$\lambda_{EDP}(e) = \int_{im=0}^{\infty} p(EDP > e|IM = im) \left| \frac{d\,\lambda_{IM}(im)}{d\,im} \right| d\,im \tag{3}$$

That is, we need to integrate over all values of IM the probability of EDP exceeding e given the intensity measure im, times the probability of the intensity measure.[2]

Verify equation 3.

1.11.6 dsedp

An `dsedp`-type `recorder` reports on the relationship between EDP and damage states.

```
recorder dsedp cgroup1 at:0.00:0.01:0.15
```

EDP	p(DS>=DS1)	p(DS>=DS2)	p(DS>=DS3)	p(DS>=DS4)
0.00	0.000000	0.000000	0.000000	0.000000
0.01	0.883974	0.018659	0.000106	0.000008
0.02	0.998294	0.363393	0.024354	0.004773
0.03	0.999960	0.746737	0.169162	0.057262
0.04	0.999998	0.916739	0.405834	0.195177
0.05	1.000000	0.973890	0.625358	0.381663
0.06	1.000000	0.991738	0.780945	0.561467
0.07	1.000000	0.997303	0.877131	0.705424
0.08	1.000000	0.999084	0.932490	0.808912
0.09	1.000000	0.999676	0.963197	0.878668
0.10	1.000000	0.999881	0.979937	0.923893
0.11	1.000000	0.999954	0.989010	0.952543
0.12	1.000000	0.999982	0.993934	0.970457
0.13	1.000000	0.999993	0.996619	0.981586
0.14	1.000000	0.999997	0.998096	0.988485
0.15	1.000000	0.999999	0.998916	0.992766

By default, `dsedp` will output a column for each damage state, but you can override this with the `cols` parameter:

```
recorder dsedp cgroup1 cols:[1, 4] at:0.00:0.01:0.15
```

EDP	p(DS>=DS1)	p(DS>=DS4)
0.00	0.000000	0.000000
0.01	0.883974	0.000008
0.02	0.998294	0.004773
0.03	0.999960	0.057262
0.04	0.999998	0.195177
0.05	1.000000	0.381663
0.06	1.000000	0.561467
0.07	1.000000	0.705424
0.08	1.000000	0.808912
0.09	1.000000	0.878668
0.10	1.000000	0.923893
0.11	1.000000	0.952543
0.12	1.000000	0.970457
0.13	1.000000	0.981586
0.14	1.000000	0.988485
0.15	1.000000	0.992766

Note that in this case the lognormal distribution paramaters (`median_X`, etc.) don't make sense, and are not allowed.

$P(DS_i|EDP = e)$ is the probability that damage state DS_i will be exceeded given an EDP of e. This is trivially calculated from the parameters of the lognormal distribution for the onset of DS_i.

[2]Note that $\lambda(IM)$ is the probabilty of the intensity measure *exceeding* IM; the derivative of this indicates how likely an intensity measure of IM is.

1.11.7 dsim

An dsim-type recorder reports on the relationship between IM and damage states.

The command:

```
recorder dsim cgroup1 at:0.0:0.5:1.5
```

would produce something like (made-up numbers):

IM	p(DS>=DS1)	p(DS>=DS2)	p(DS>=DS3)	p(DS>=DS4)
0.00	0.000000	0.000000	0.000000	0.000000
0.05	1.000000	0.973890	0.625358	0.381663
0.10	1.000000	0.999881	0.979937	0.923893
0.15	1.000000	0.999999	0.998916	0.992766

As with dsedp, dsim will output a column for each damage state by default; this can be overridden with the cols parameter.

The probability of exceeding a given DS state DS_i is

$$p(DS_i|IM = im) = \int_{edp=0}^{\infty} p(DS_i|EDP = edp) \left| \frac{dp(EDP > edp|IM = im)}{d\,edp} \right| d\,edp \tag{4}$$

Verify equation 4.

1.11.8 dsrate

An dsrate-type recorder reports on the overall probability of each damage state being exceeded. This produces one value per damage state (made-up numbers):

The command:

```
recorder dsrate cgroup1
```

will produce something like this (made-up numbers):

DS1	DS2	DS3	DS4
0.10	0.08	0.04	0.01

Note that dsrate does not accept the at parameter.

The rate of DS_i is:

$$\lambda_{DS}(DS_i) = \int_{im=0}^{\infty} p(DS_i|IM = im) \left| \frac{\lambda_{im}(im)}{d\,im} \right| d\,im \tag{5}$$

Verify equation 5.

1.11.9 lossds

A lossds-type recorder reports on the probability the loss will exceed a given value for each damage state.

The command:

```
recorder lossds cgroup1 at:[1E3, 1E4, 1E5]
```

would produce something like this (made-up numbers):

Loss	DS1	DS2	DS3	DS4
1E3	0.90	0.70	0.50	0.20
1E4	0.80	0.40	0.30	0.10
1E5	0.70	0.30	0.20	0.05

How is this calculated?

1.11.10 lossedp

A lossedp-type recorder reports on the relationship between EDP and expected loss.

```
recorder lossedp cgroup1 at:0.0:0.05:0.15
```

would produce something like (made-up numbers):

EDP	μ(LOSS EDP)	σ(LOSS EDP)
0.00	0	0
0.05	5.0E5	5.0E5
0.10	9.0E6	4.2E5
0.15	1.0E6	4.0E5

How is this calculated?

1.11.11 lossim

A lossim-type recorder reports on the relationship between IM and expected loss.

```
recorder lossim cgroup1 at:0.0:0.05:0.15
```

would produce something like (made-up numbers):

How is this calculated?

IM	μ(LOSS | IM)	σ(LOSS | IM)
0.00	0	0
0.05	5.0E5	5.0E5
0.10	9.0E6	4.2E5
0.15	1.0E6	4.0E5

1.11.12 Python Code

Python code can be evaluated to produce parameters. The code is delimited by `$(` and `)`. For example, in the command:

```
recorder detfn im_func
         at:$(numpy.arange(0.00, 0.05, 0.01))
```

The Python expression evaluates to the array: Note that this requires the `numpy` library be loaded, which slat does by default.

1.12 Variables

1.12.1 Setting

We can define local variables with the SET command. The expression after the = will be evaluated by Python:

```
set probabilities=[0.16, 0.50, 0.84]
```

The type of the variable will depend on the Python expression. In the example above, `probabilities` will be a list of numbers. In contrast,

```
set FILE_ROOT="slat_output"
```

will set `file_root` to a string.

1.12.2 Referencing

To use a variable, prefix it with a dollar sign($). The command:

```
recorder probfn edp_fn cols:$PROBABILITIES at:0.0:0.1:0.5
         create:$FILE_ROOT
```

sets up an `probfn` recorder with colums of 16%, 50%, and 84%, to be saved to the file "slat_output".

I can see users wanting to set up a base string for a file name, and add to that for each output file. For example, writing the EDPIM recorder to ***slat_output_EDPIM****, and the EDPRATE recorder to* ***slat_output_EDPRATE****. We need to identify a syntax for this.*

Note that $probabilities is different from $(probabilities)$. The first will look up the value in a local table of things that have been 'SET'; the latter will exececute Python code directly (using the eval() function) to evaluate the variable.

Also note that eval() will use the builtins namespace, for safety.

Can we use local variables in code?

```
set local_var=0
set local_var=$($local_var + 1)
```

I think we can, if we substitute the variable(s) before executing the code.

1.13 analyze

The analyze command causes slat to generate the data for each recorder, performing (and caching) the necessary calculations as it does does so.

```
# Perform the analysis:
analyze

# Perform the analysis again--calculations should be
# faster, as results have been cached:
analyze
```

Changing the integration parameters will invalidate the cache. analyze will need to re-evaluate all calculations with the new integration parameters.

```
integration maq 1E-6 2048
analyze
```

Each time we run analyze the output files are overwritten, which is probably not what we would like. The easiest way to prevent this may be to change the output directory.

```
option outputdir "results"
analyze  # Output files are in the 'results' subdirectory

option outputdir "more_results"
analyze  # Output files are in the 'more_results'
         # subdirectory
```

Harder, would be to support some sort of dynamic naming–interpret a pattern when the recorder is actually run. Something like:

```
set prefix="first"
recorder detfn im_func 0.0:1.0:0.001
         create:$prefix + "-im_func.txt"
analyze  # Data is written to 'first-im_func.txt'
```

```
set prefix="second"
analyze  # Data is written to 'second-im_func.txt'
```

It might be better to have specific syntax for deferred expansion, to avoid confusion.

1.14 Redefining

What happens if we try to redefine an identifier?

```
detfn hyperbolic im_func [1221, 29.8, 62.2]
im  im1  im_func
detfn hyperbolic im_func [122.1, 29.8, 62.2]
```

I think the second `detfn` should replace the first, including in `im1`, and invalidate any caches in `im1`, or functions that rely (however indirectly) on `im1`.

Similarly, if we redefine im1:

```
im im1 <some other function>
```

...I would expect `im1`'s cache (if any) to be cleared, along with all functions that rely on `im1`.

2 Lognormal Distributions

`SLAT` makes extensive use of lognormal distributions. If X is a lognormal distribution (and therefore $\ln(X)$ is a normal distribution), it can be described by mean (μ) and standard deviation (σ) of $\ln(X)$. However, there are other parameters which some users may prefer. Table 1 lists those `SLAT` will support:

Table 1: Lognormal distribution paramers supported by `SLAT`.

Symbol	Identifier	Meaning
μ_{lnX}	`mu_lnX`	Mean of the normal distribution $\ln(X)$
median_X	`median_X`	Median of the lognormal distribution X
mean_X	`mean_X`	Mean of the lognormal distribution X
$\sigma_{\ln X}$	`sigma_lnX`	Standard deviation of the normal distribution $\ln(X)$
sd_X	`sd_X`	Standard deviation of the lognormal distribution X

To describe a lognormal distribution, the user must provide one of `mu_lnX`, `median_X` or `mean_X`, along with either `sigma_lnX` or `sd_X`. The first parameter controls the placement of the distribution along the X axis; the second controls the spread of the distribution. Not all combinations of parameters will be supported (see Table **??**).

The relationship between $\mu_{\ln\{X\}}$ and median_X is straightforward logarithmic one:

$$median_X = e^{\mu_{\ln X}}$$

and

$$\mu_{\ln X} = \ln median_X$$

In code, this would look like:

```
median_X = exp(mu_lnX);
mu_lnX = log(median_X);
```

The relationship between $\mu_{\ln\{X\}}$ and mean_X is affected by $\sigma_{\ln\{X\}}$:

$$mean_X = e^{\mu_{ln X} + \frac{\sigma^2_{\ln X}}{2}}$$

and

$$\mu_{\ln X} = \ln\left(mean_X\right) - \frac{\sigma^2_{\ln X}}{2}$$

In code, this would look like:

```
mean_X = exp(mu_lnX + sigma_lnX * sigma_lnX / 2);
mu_lnX = log(mean_X) - sigma_lnX * sigma_lnX / 2;
```

The relationship between $\sigma_{\ln X}$ and sd_X is the most complex:

$$sd_X = mean_X \sqrt{e^{\sigma^2_{\ln X}} - 1}$$

and

$$\sigma_{\ln X} = \sqrt{\ln \frac{sd_X^2}{mean_X^2} + 1}$$

In code, this would look like:

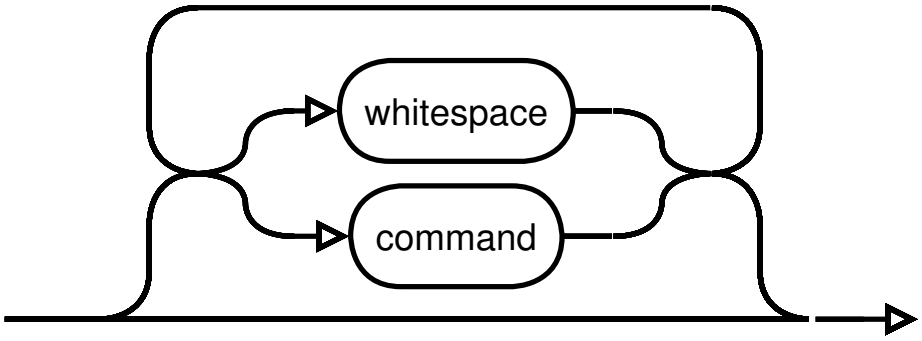
```
sd_X = mean_X * sqrt(exp(sigma_lnX * sigma_lnX) - 1);
sigma_lnX = sqrt(log(1 + (sd_X*sd_X) / (mean_X*mean_X)));
```

`SLAT` will store lognormal distributions as `mu_lnX` and `sigma_lnX`; if the user provides other parameters the will be converted. Table **??** shows how different combinations are converted; note that `sd_X` is only supported in conjunction with `mean_X`.

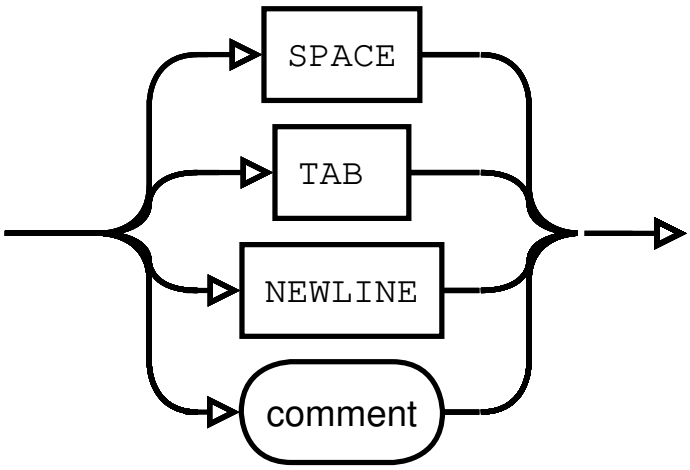
Placement	Spread	Action
mu_lnX	sigma_lnX	Nothing
mu_lnX	sd_X	Not supported
median_lnX	sigma_lnX	Convert median_lnX to mu_lnX.
median_lnX	sd_X	Not supported
mean_X	sigma_lnX	Use mean_X and sigma_lnX to calculate mu_lnX.
mean_X	sd_X	Use mean_X and sd_X to calculate sigma_lnX, then use sigma_lnX and mean_X to calculate mu_lnX.

3 Formal Definition

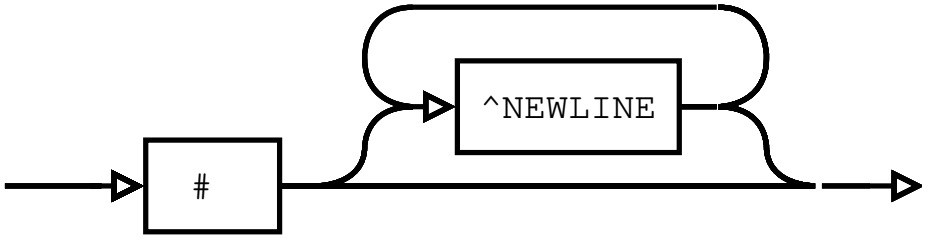
script :



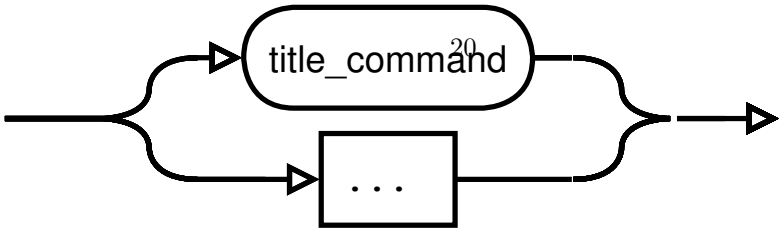
whitespace :



comment :

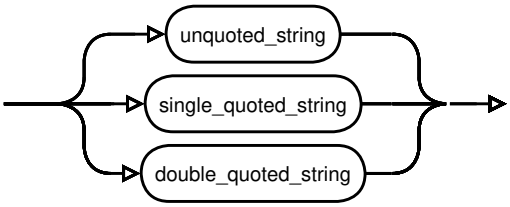


command :



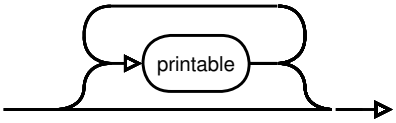
title_command :

string :

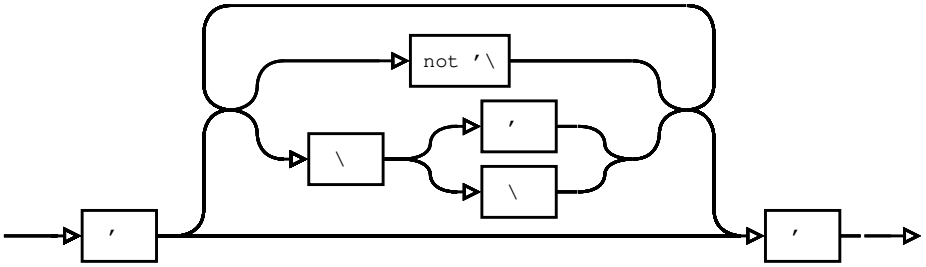


printable :

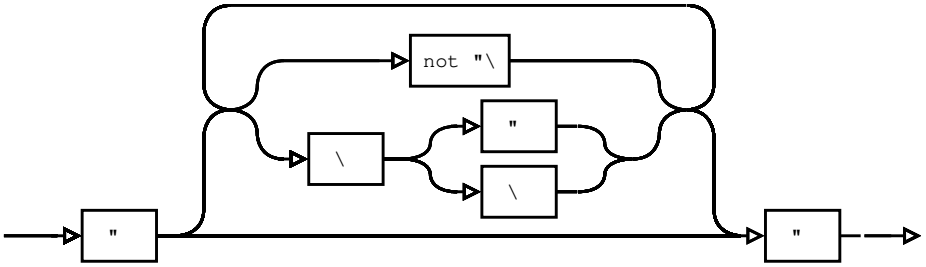
unquoted_string :



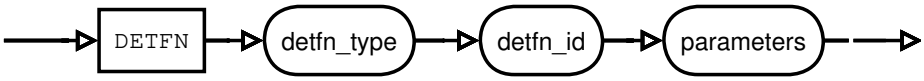
single_quoted_string :



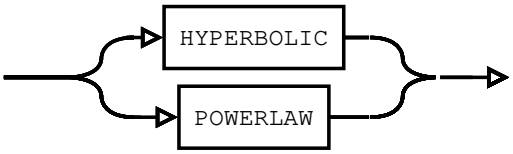
double_quoted_string :



detfn_cmd :

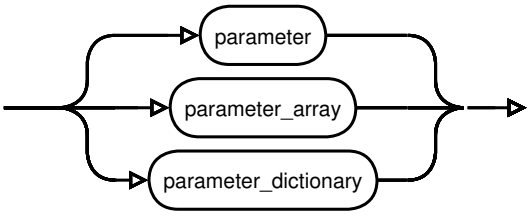


detfn_type :

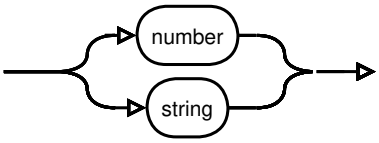


detfn_id :

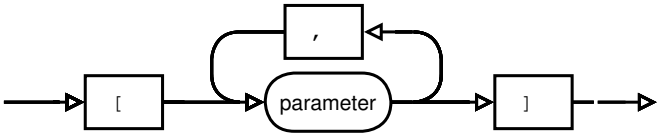
parameters :



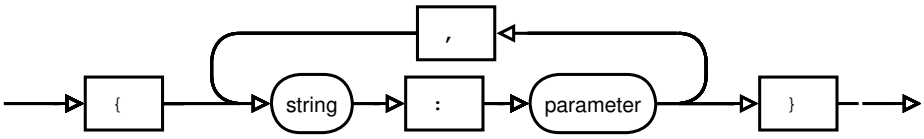
parameter :



parameter_array :



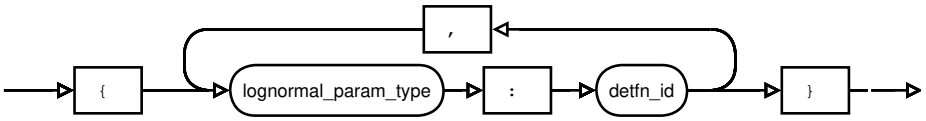
parameter_dictionary :



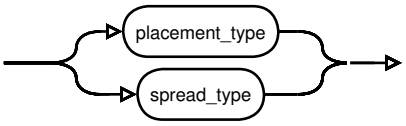
probfn_cmd :

probfn_id :

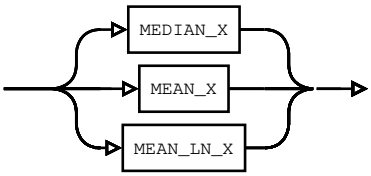
lognormal_params :



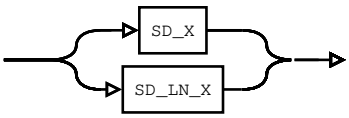
lognormal_param_type :



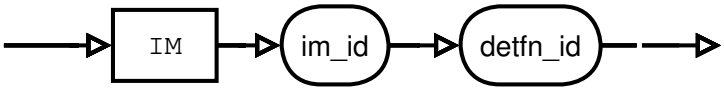
placement_type :



spread_type :

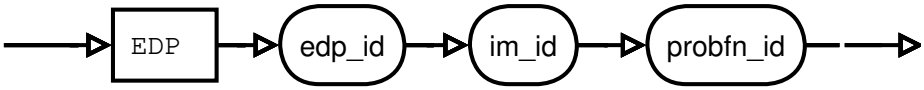


im_rel :



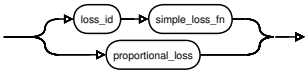
im_id :

edp_rel :

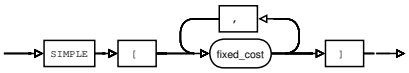


edp_id :

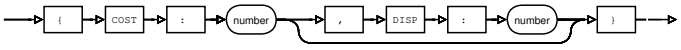
lossfn :



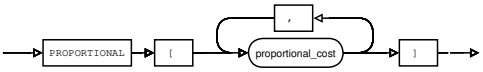
simple_loss_fn :



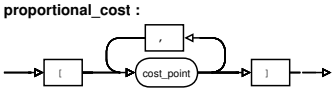
fixed_cost :



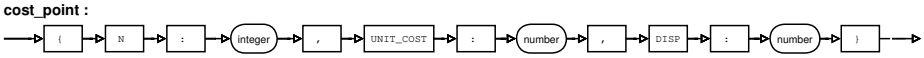
proportional_loss :



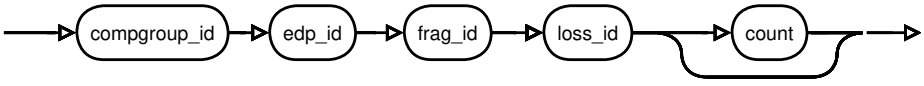
proportional_cost :



cost_point :



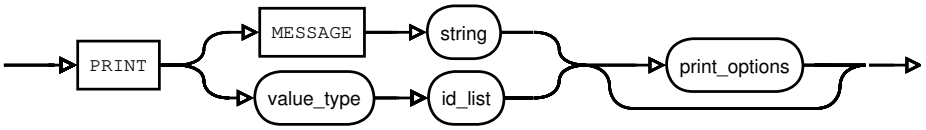
compgroup :



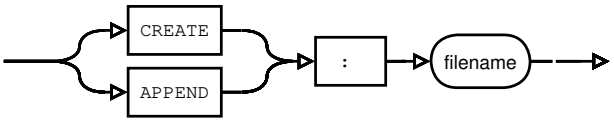
compgroup_id :

count :

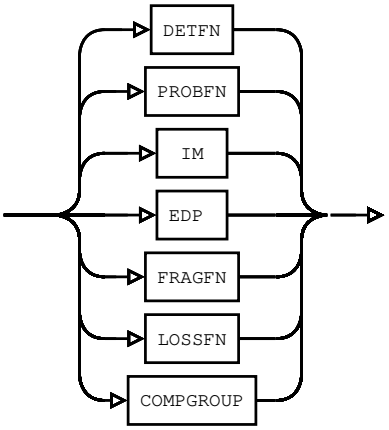
print_cmd :



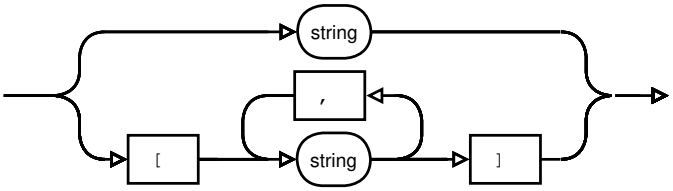
print_options :



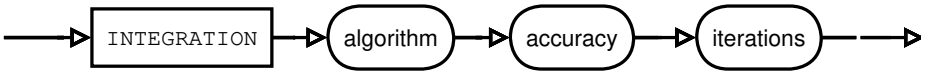
value_type :



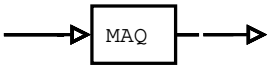
id_list :



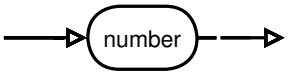
integration_cmd :



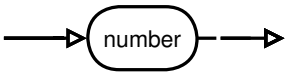
algorithm :



accuracy :

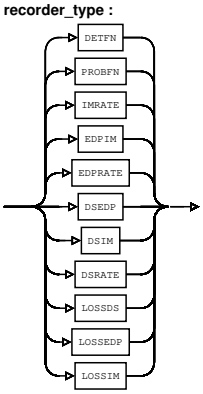


iterations :

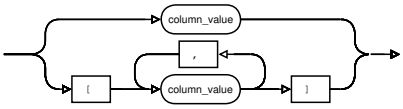


recorder_cmd :

recorder_type :

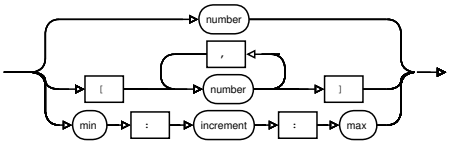


column_values :



column_value :

at_values :



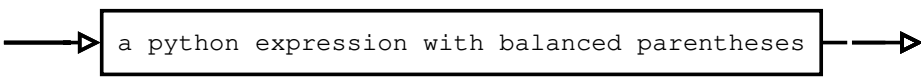
min :

increment :

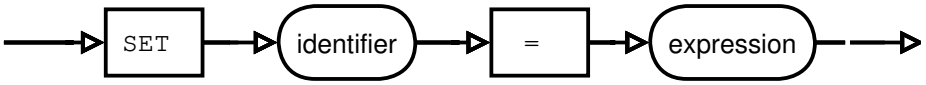
max :

python_code :

python_expression :

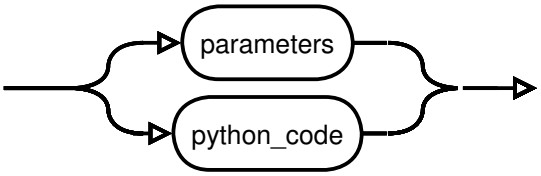


var_set_cmd :

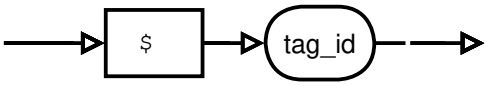


identifier :

expression :



var_ref :



analyze_cmd :