Installing and Using Python LISA

Version 1.00, 08/27/07

University of Illinois

Table of Contents

System Requirements and Installation

System Requirements:

Macintosh Operating System 10.3 or later. (The model will probably also run under Windows if you have PyGame and Python but I can't tell you how to set it up.)

Python 2.4 (for Macintosh; note that Python 2.5 is the most recent version, but it does not support PyGame)

PyGame (a set of graphical routines for Python, included with this package)

Installation:

1) **Install Python 2.4**: go to http://www.python.org/ or http://pythonmac.org/packages/py24-fat/index.html

2) **Install PyGame** (included in this package in the folder `PygameInstalls`; there are a total of three applications; install them in the order listed in the folder; you may or may not need the third one)

3) **Place the folder `PyLISA` into:**

`HardDrive:/library/frameworks/python.framework/versions/2.4/`

where "*`HardDrive:`*" is a variable to be replaced by the name of your hard drive; the rest of the path name is literal.

About this Manual

This instruction manual was written under the assumption that the person reading it has at least some familiarity with the LISA model. It therefore uses some LISA-specific jargon (e.g., "mapping connection", "self-supervised learning", "SP", etc.) without defining it. If you are not familiar with LISA (e.g., if any of the preceding jargon surprised or confused you), please see Hummel & Holyoak (2003) in *Cognitive Studies* for an overview or Hummel & Holyoak (2003) in *Psychological Review* for all the gory details. Both papers can be found at: http://www.psych.uiuc.edu/~jehummel/publications.php

Note also that this manual was written for the version of LISA written in Python by John Hummel. These instructions will *not* work for the version of LISA written in Python by Derek Devnich.

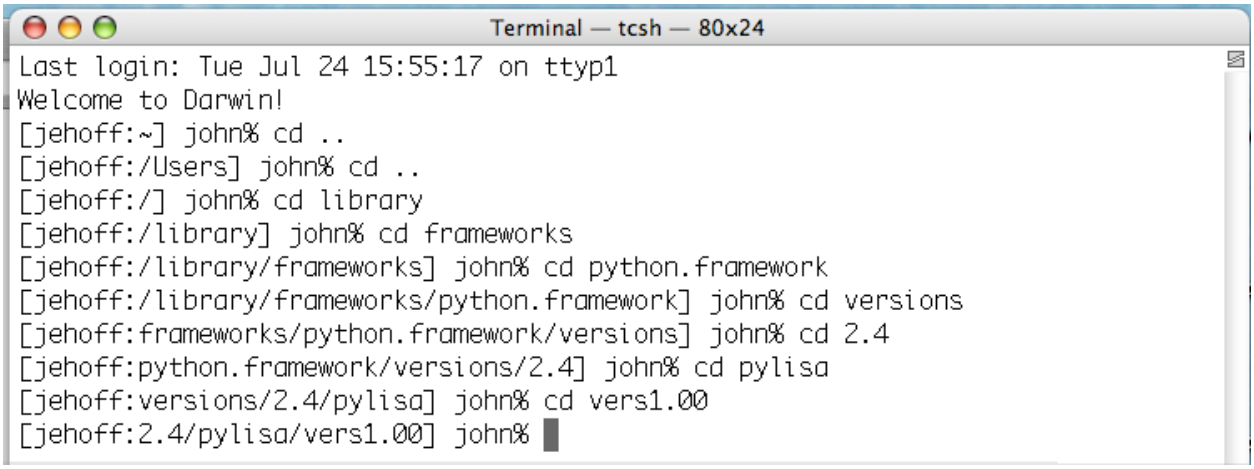
Running LISA

1) **Launch Terminal** (you can find it in your Applications folder)

2) **Go to :**
   `HardDrive:/library/frameworks/python.framework/versions/2.4/PyLISA/Vers1.00/`

where "*`HardDrive`*`:`" is again a variable to be replaced by the name of your hard drive; the rest of the path name is literal.

Your Terminal window will now look something like this:



3) At the prompt (%) enter: `python lisa.py`

Now your Terminal window should look something like this…

```
* * * * python LISA * * * *
* * * Version  1.00   * * *

   last modified 7/24/07

------- Main Menu -------

(F)ile:  <none>
(P)ath: Data/
(M)odify parameters
(R)un
(B)atch run
(V)iew network
(E)mpty memory
(S)creen size...
Show (N)otes
(G)o: Run  lovetri/lovetri7 blind

(Q)uit

LISA>
```

… *except* that it will initially be occluded by a large black window entitled "pygame window". Click on the Terminal window to bring it to the front and you should see something like the image above.

The **Terminal window** is where you access the LISA Main Menu and its submenus.

The **pygame window** is where the network is displayed during a run, and where you issue various directives (such as quitting, speeding up or slowing down the graphical display, etc.) during a run of the model.

The LISA Main Menu

All file options are shown in parentheses next to (or embedded in) the option name. For example `(F)ile` indicates that pressing the `F` key, followed by a carriage return, activates the File option in the Main Menu.

`File (F)` allows you to specify a simulation (.SYM) file name. (SYM files are text files that tell the code how to construct and run a simulation.)

For example, to choose the file lovetri7.sym, you would enter `F` <cr> at the `LISA>` prompt (i.e., Main Menu) and then enter `lovetri7` at the prompt `file name (omit suffix):`

As the prompt indicates it is unnecessary (indeed, bad) to explicitly specify the `.sym` suffix. (The code expects that suffix as a universal and so adds it itself.)

Once there is a file in memory its name is displayed next to the `(F)ile` option in the Main Menu, e.g., `(F)ile: lovetri7`

`Path (P)` allows to specify a subdirectory path for reading `.sym` files and for writing output (`.run` and `.bat`) files.

The code looks for all sym files in a folder called `Data/` inside the main LISA folder (i.e., `Vers1.00/` as of this writing). It also saves all output files to `Data/`. However, within `Data/` it is possible to create folders to house families of sym files. Several such folders are included in this package.

The current subdirectory path is specified next to the `(P)ath` option in the Main Menu, e.g., `(P)ath: Data/lovetri/`

The code will save any output files to the same folder containing the sym file that generated the simulation.

Subdirectories can be nested to arbitrary depth but they must all be located inside `Data/`.

`Modify parameters (M)` allows you to modify various parameters that control the model's operation. After entering `M<cr>` at the `LISA>` prompt, you will be asked to choose one of three sub-menus:

`(F)ile parameters` allows you to modify parameters governing which kinds of details are saved to the output (`.run` or `.bat`) file. I do not detail those parameters here because they are fairly intuitive based on their names. The one that may not be intuitive based on its name is `Min Vertical Wt. to Save`. This parameter allows you to set the threshold on saving inferred object and predicate semantic units to file: how large the weight from a semantic unit to an inferred object or predicate unit must be in order to write the semantic unit's name and weight to the output file.

`(R)un parameters` allows you to modify various parameters governing the model's operation. For more detail, see the section on Modifying Parameters.

`(D)angerous parameters` are those that govern the most basic nuts and bolts of model's behavior (primarily excitatory and inhibitory interactions between classes of units) and which I do not recommend changing. Most of them have sweeping, cascading and therefore largely unpredictable effects. Most of these effects can be achieved more straightforwardly by modifying various parameters in the `(R)un`

`parameters` menu. For more details on the Dangerous Parameters, see the section on Modifying Parameters.

`Run (R)` runs the model on whatever `.sym` file is currently in memory. (If there is a file in memory, its name will appear next to the `(F)ile:` option in the Main Menu.) The output of a single run is saved to a `.run` file whose file name is otherwise the same as the name of the `.sym` file that generated the simulation. During a run, text output is written to the Terminal window and, if desired, the network is displayed graphically in the pygame window.

After you issue the `(R)un` command in the Main Menu, a submenu will appear in the Terminal window asking whether you want a `(g)raphic` or `(b)lind` [i.e., sans graphics] run. See Interpreting the Model's Output for more detail.

`Batch run (B)` allows you to run a large number of simulations in batch (and without graphical output, to maximize speed), saving the results to a `.bat` file. When you hit `B` <cr> you will be taken to the Batch Run menu, which looks something like this:

```
Current Batch Sequence:

Current Path: Data/
(p) Change path
(n) New batch element
(d) Delete batch element
(r) Run batch sequence
(a) Abort batch sequence
```

The purpose of this menu is to allow you to define a sequence of simulations to run in batch: Which sym files to run, where to find them and how many times to run each one.

The `Change path` command (`p`) allows you to tell LISA where to find the sym files.

The `New batch element` command (`n`) allows you to tell LISA to run a sym file that resides inside the `Current Path`. After you issue this command, you will be asked to specify (a) a sym file name (give the file name without the .sym suffix) and (b) how many times to run the simulation specified in the file. (Since LISA has numerous stochastic components, it is often useful to run a simulation more than once.) Note that you can change the path several times within a single batch sequence, running sym files from a variety of different folders. The list of batch elements you have defined will appear under the "`Current Batch Sequence:`" header above the menu.

`Delete batch element` (`d`) allows remove an element from the batch sequence (e.g., because it contains an error, or for some other reason, you have decided you do not wish to run it).

`Run batch sequence (r)` runs the files specified in the sequence the specified number of times each. Before the run commences you will be asked to provide a file name for saving the results of the batch simulations. Do not include the suffix .bat; LISA adds it automatically. Note that if you want to specify a particular part for

saving the .bat file, then you must include it as part of the name (e.g., "`lovetri/lovetri7`").

`Abort batch sequence (a)` aborts the run.

`View network (V)` displays all the nodes in the network, along with their interconnections and the sequence of events specified in the sym file, in the Terminal window. This function is useful for debugging sym files because it tells you what The Code thinks the sym file says (which may or may not be the same as what You think it says).

`Empty memory (E)` disposes of all units, connections, etc. and will initializes the graphical display. Doing so allows you to load and run one sym file, and then load and run a second sym file without having to quit LISA in between. (If you load a second sym file without emptying memory in between, the second network will be built on top of the first.) As of this writing, this function is still *potentially* buggy… which is to say it *seems* to work as best I can tell, but there may be surprises. Please email me at the address at the end of this document if you discover a bug.

`Screen size (S)` allows you to resize the pygame window.

`Show Notes (N)` displays the latest research/development notes, including problems that remain to be solved, etc. I apologize in advance for any profanity contained therein.

`Go: `*`path`*`/`*`filename`*` with/without graphics (G)` sets the Path to *`path`*, the File Name to *`filename`* and then runs the network either with or without graphics (as specified in the string following `Go:`). The value of `path`, `filename` and `graphics/no-graphics` are specified near the beginning of LISA.py (the master file for the Python implementation of LISA). For example:

```
go_path  = 'lovetri/'
go_file_name = 'lovetri8'
go_use_graphics = True
```

would run lovetri/lovetri8.sym with graphics turned on. The `(G)o` option is a convenient way to run a simulation with a single command rather than having to enter the path etc. every time.

`Quit (Q)` quits LISA, returning you to the Terminal.

Runtime Directives: It is also possible to issue a few directives in the pygame window during a run. In contrast to the menu items in the LISA Main Menu (in the Terminal), it is not necessary to follow these directives with a carriage return:

s toggles between the "speed" and "step" modes of updating. In "speed" mode, LISA repeatedly updates the state of the network and the graphical display without user input. In "step" mode, the model updates only a few iterations per button-press (or mouse movement) by the user. Step mode is useful for watching the model work in excruciating detail.

- (the minus key) slows the simulation down by increasing the rate at which the graphical display is updated. Each successive press doubles the graphic refresh rate (i.e., cuts in half the number of network updates [iterations] per graphical update). At the slowest speed, the graphical display is updated every iteration.

= (the equals key) speeds the simulation up by decreasing the rate at which the graphical display is updated. Each successive press halves the graphic refresh rate (i.e., doubles the number of iterations that go by between graphical updates).

q quits the current simulation, returning you to the Main Menu in the Terminal (it is necessary to click on the Terminal to bring it to the front).

esc quits the current simulation and also quits out of LISA.

←→↑↓ (the arrow keys) allow you scroll the display window in the corresponding direction. Note that the Window, not the items displayed therein, follows the direction of the arrow. For example, the UP arrow moves the window view up, and hence the objects in the window down.

Writing Simulations (SYM files)

Simulation files are plain text files with the .sym suffix. (Read: Do ***not*** write them in Word and save them as Word files. They ***must*** be *plain* text.) I have included several examples in the `Data/` folder, including `Data/syntax.sym`, which illustrates many (but not all) of the sym file syntax conventions. Checking out other included simulations will reveal other features.

A sym file tells LISA how to construct a network (i.e., which propositions to build in which analogs, which semantics to connect to which objects and predicates, etc.) and what to do with that network once it's built.

`build.py` (henceforth simply "Build") is the simulation parser, i.e., the code that reads the sym file and constructs the network and the simulation sequence. In the following, I will use both upper- and lower-case letters to describe commands, etc., for clarity. However, Build is not case-sensitive: everything except the text following `Note:` (see below) is immediately converted to all uppercase. Thus, it is not necessary to follow the case conventions used in the commands below. Similarly, in the sample sym files, commands are indented systematically (e.g., all commands having to do with a given analog are indented under the command that initiates the creation of the analog). These conventions are strictly for readability and are not required by Build.

I recommend that you open one of those sample files and refer to it as a concrete example as you read these instructions.

> **A NOTE ABOUT PUNCTUATION IN SYM FILES**: I am not a professional programmer, and I am *not* good at writing text parsers. For this reason, in your sym files ***all necessary punctuation must be separated from any commands or unit names by spaces*** (one space is sufficient). That is, ***punctuation must not touch text***. Punctuation that touches text is interpreted, not as punctuation, but as part of the text. I'm sorry about that, but it's just the way it is. When you start writing sym files you *will* forget this and Build will crash when it fails to find the punctuation you thought you had put there. I apologize in advance that this will happen to you. But you'll survive it. And you can't say I didn't warn you.

Sym files are organized hierarchically. Build accepts six commands at the top of the hierarchy.

Top-Level Commands:

`Note:` designates text to be copied from the sym file to the output (.run or .bat) file. All text following the `Note:` command (on the same line) is copied verbatim into the output file. Use it to write notes to yourself describing the sym file (e.g., "This is my attempt to simulate…" or "This is the same as bla bla except for the value of the bla bla parameter…"). (Note that in the case of `Note:`, the colon is not punctuation; it is part of the command. That's why you can (indeed, must) run it up against the word Note.)

`Parameters` allows you to set runtime and file-saving parameter values in the sym file (i.e., so that you don't have to do it from the Main Menu). Any parameter you do not set in the sym file will simply adopt its default value (or the value you set from the `(M)odify parameters` option in the Main Menu). Note that any runtime parameter will adopt the last value to which it was set: If you set the parameter in the `(M)odify parameters` menu *and then* read the sym file, then the model will run with the value set in the sym file; if you read the sym file *and then* modify the parameter in the `(M)odify parameters` menu, it will run with the value set in the menu. Once you issue the `Parameters` command in the sym file, you can name the parameter(s) you wish to set, each followed by a parameter value. You must tell Build you are done modifying parameters by issuing the command `Done`. The syntax for changing a parameter value in the sym file is:

```
Parameters
   <parameter1_name>  <parameter1_value>
   <parameter2_name>  <parameter2_value>
   …
   <parameternn_name>  <parameternn_value>
Done
```

The commands (parameter names) you can issue are listed below, along with their default and legal values. A legal value of "0…1 (+)" means "any positive value is logically possible, but values greater than 1 are weird and may have weird effects". For an explanation of what each parameter does, see the section Modifying Parameters.

These commands set runtime parameters an thus affect the model's behavior:

Command/Parameter:	Default Value:	Legal Values:
`UnlimitedWM`	`False`	`True/False`
`SemanticNoise`	`0`	`0…1 (+)`
`SemanticDeath`	`0`	`0…1`
`Attention`	`1`	`0…1 (+)`
`DriverInhibition`	`1`	`0…1 (+)`
`RecipInhibition`	`1`	`0…1 (+)`
`HebbLearningRate`	`1`	`0…1 (+)`
`BailUponSettling`	`False`	`True/False`
`MappingAalgorithm`	`VERS142`	`VERS142/H&H9703`
`WithinGroupSupport`	`1`	`-∞…∞`

These parameters determine what does and does not get saved to the output (`.run` or `.bat`) file at the end of a run:

Command/Parameter:	Default Value:	Legal Values:
`SaveGroupHebbs`	`True`	`True/False`
`SavePropHebbs`	`True`	`True/False`

```
SaveSPHebbs              True      True/False
SaveOPHebbs              True      True/False
```

{ (a left curly-brace) designates a comment. Build ignores all text following a { (on the same line). In contrast to notes, comments are not copied into the output file.

Analog <analog_name> tells Build to start constructing a new analog with the name <analog_name>. For example in response to `Analog Source` Build will create a new analog object with the name "Source". Propositions, objects, predicates and groups (i.e., the stuff that the units in LISA represent) all live inside specific analogs, so the `Analog` command must be issued before these units can be defined (as elaborated below).

Done tells Build that the definition of the current analog (or the simulation sequence, as described below) is complete. If your sym file has issued the `Analog` command (followed by definitions of the units in an analog), then `Done` *must* be the next top-level command issued (otherwise Build will either report an error or crash or both). In other words, your sym file must finish defining one analog before moving on to define the next, or defining the simulation sequence.

Sequence tells Build to start defining the simulation sequence, i.e., the set of instructions telling the model what to fire when, when to update the mapping connections, etc.

Analog Construction Commands:

After you issue the `Analog <analog_name>` command, Build expects you to issue either the `Done` command (ending the definition of the analog) or one of four high-level analog definition commands:

DefPreds tells Build to start defining the units that will represent the roles of the predicates in the current Analog. Once DefPreds is issued, Build expect the next thing it encounters (at the top level within DefPreds) to be either the word `end` or the name of a predicate. In other words, if Build is not in the middle of defining a predicate (as elaborated below) and it encounters the word `end`, then it will think it is done defining predicates in the current Analog. An unfortunate consequence of this convention is that you cannot name a predicate "end". (If you need such a predicate, try something related but different, such as "the_end".) Each predicate definition ends with a semicolon. There are two ways to define predicates: *auto-coding* and *hand-coding*.

Auto-coding a predicate: Auto-coding is easier and than hand-coding but gives you less control over the semantic coding of each role of the predicate. To auto-code a predicate, write the name of the predicate, followed by the number, *n*, of roles, followed by the name of each semantic unit to be attached to the roles. Build will create *n* predicate (role) units, numbering each according to its place (first, second, etc.), create *n* copies of each named semantic feature, numbering each

according to its place, and connect the predicate (role) units to the corresponding semantic units.

For example:

```
DefPreds
     Loves 2 strong positive emotion loves ;
End
```

will create two predicate (role) units:

LOVES1 connected to the semantics STRONG1, POSITIVE1, EMOTION1, LOVES1.

and

LOVES2 connected to the semantics STRONG2, POSITIVE2, EMOTION2, LOVES2

Note that the predicate definition ends with a semicolon. (And note that the semicolon does not touch the text!)

Be aware that auto-coding a predicate results in predicate roles with non-overlapping semantic units. For example, the semantic units connected to LOVES1 above are completely separate from those connected to LOVES2.

Hand-coding a predicate: Hand-coding allows you to specify exactly which semantics are connected to each role of a predicate. To hand-code a predicate, specify the predicate's name, skip the number, *n*, of roles (or use 0), and then specify the semantics defining each role inside square brackets.

For example:

```
DefPreds
     Loves [ strong1 positive1 emotion-agent loves1 ]
           [ emotionpatient loves2 beloved ] ;
End
```

will create two role units:

LOVES1 connected to STRONG1, POSITIVE1, EMOTION-AGENT, LOVES1

and

LOVES2 connected to EMOTIONPATIENT, LOVES2, BELOVED

Note that each role is closed with a right square bracket and the predicate definition ends with a semicolon. (And note that the punctuation does not touch the text!)

`end` if encountered outside the definition of a predicate (i.e., after a semicolon), ends the definition of predicates in the current Analog.

`DefObjs` tells Build to start defining the units that will represent the objects in the current Analog. Once DefObjs is issued, Build expect the next thing it encounters (at the top level within DefObjs) to be either the word `end` or the name of an object unit. In other words, if Build is not in the middle of defining an object (as elaborated below) and it encounters the word `end`, then it will think it is done defining objects in the current Analog. As with predicates, an unfortunate consequence of this convention is that you cannot name an object "end". Each object definition ends with a semicolon. The first word in each object definition is interpreted as the name of the object unit and every word after that (preceding the semicolon) is interpreted as the name of a semantic unit connected to that object. For example:

```
DefObjs
     John human adult male john ;
     Mary human adult female mary ;
End
```

will create two object units:

JOHN connected to the semantics HUMAN ADULT MALE JOHN

and

MARY connected to the semantics HUMAN ADULT FEMALE MARY

`DefProps` tells Build to start defining propositions in the current Analog. Note that you must define all the predicates and objects to which an analog's propositions will refer *before* defining the propositions themselves. Each proposition definition starts with the name of the proposition itself, followed by the name of the predicate, an open parenthesis, the names of all the proposition's arguments (which can either be objects or propositions you have already defined), a close parenthesis, and finally a semicolon. Note that the number of arguments must agree with the number of places in the predicate as you defined it in `DefPreds`. Indicate that you are done defining propositions with the command `End`. For example:

```
DefProps
   P1 Loves ( John Mary ) ;
End
```

constructs the proposition *loves* (John, Mary) in LISAese. It will consist of one P unit (P1) and two SPs (SP1.1 and SP1.2). SP1.1 will connect to LOVES1 (constructed in `DefPreds`) and JOHN (constructed in `DefObjs`) and SP1.2 will connect to LOVES2 and MARY. Each of these predicates and objects will have the semantic

features assigned to them in `DefPreds` and `DefObjs`, respectively. (Note how the words in the above syntax do not touch the necessary punctuation: ( ) and ;.)

By default, Build assigns each proposition and importance value (i.e., *pragmatic centrality*; Holyoak & Thagard, 1989) of 1.0. An optional command, not shown in the example above, allows you to define a proposition's importance to a non-default value in the sym file. To do so, simply state the importance after the close paren and before the semicolon. For example:

```
DefProps
   P1 Loves ( John Mary ) 10 ;
End
```

gives P1 an importance value of 10. There is no necessary bound on a proposition's importance (i.e., they can be as large or small as you wish), but assigning negative importance values is not advised as it may result in a division by zero error. (The algorithm for random firing divides a quantity by a function that includes a sum of importances, which can in principle go to zero if some of those importances are negative.)

If one proposition takes another as an argument, then the argument proposition must be defined before it is referred to by another proposition. For example, as long as "knows" and "Bill" are defined (in `DefPreds` and `DefObjs`, respectively), then:

```
DefProps
   P1 Loves ( John Mary ) ;
   P2 Knows ( Bill P1 ) ;
End
```

will result in a perfectly acceptable hierarchical proposition stating that Bill knows that John loves Mary: *knows* (Bill, *loves* (John, Mary)). By contrast:

```
DefProps
   P1 Knows ( Bill P2 ) ;
   P2 Loves ( John Mary ) ;
End
```

will result in a run-time error.

`DefGroups` tells Build to start defining *groups* of propositions (and of other groups). Groups are units that allow you to group propositions into meaningful sets within an analog (e.g., into causal chains, etc.). They are a new addition to LISA and we have not yet fully worked out the theory of how they should work. As a result, Build probably allows you to do some things that you (or we) may later discover are ill-advised. (For example, you can currently connect a group both to propositions and to other groups. I suspect it may be better to define groups in a strict hierarchy so that a

given group connects either to propositions or to lower-level groups but not both. We shall see.)

After you issue the `DefGroups` command, each group definition begins with the name of the group. Within a group's definition, it is possible to connect the group to one or more propositions (the `Props:` command), one or more other groups (the `Groups:` command) and one or more group semantics (the `Semantics:` command). Each connection command ends with the `%` command, the definition of a complete group ends with the `%%` command, and the end of `DefGroups` as a whole ends with the `end` command. (In the following, recall that `{` denotes a comment so that text following `{` is ignored by Build.)

For example:

```
DefGroups
    G1
        Props: P1 P2 %
        Semantics: cause %
        %% { ends the group def of G1
    G2
        Props: P3 %
        Semantics: effect %
        %% { ends def of G2
    G3
        Groups: G1 G2 %
        Semantics: cause-relation %
        %%
end { def groups
```

defines three groups: G1 is connected to proposition units P1 and P2 and to the group semantic "cause". (Group semantics are completely separate from object and predicate semantics. Thus, the group semantic "cause" is a separate unit from any predicate or object semantic unit called "cause".) G2 is connected to the proposition P3 and the group semantic "effect". G3 is connected to groups G1 and G2 and to the group semantic "cause-relation". Together, groups G1…G3 represent the idea that P1 and P2 jointly cause P3. (This knowledge can also be represented more explicitly using propositions; this group-based representation is a quasi-explicit representation of the causal relation.) (Why I chose to end the parts of a group definition with a `%` rather than a `;` I do not recall. This convention may change in future versions of the code.)

You will find that one of the most common uses of groups is to define causal relations, as in the above example. I recommend using the general structure above to represent causal groups, with one group for the cause, another for the effect and a third to link the cause to its effect. A much more efficient way to generate the structure above is to issue the `Cause` command inside `DefGroups`:

```
DefGroups
   Cause ( P1 P2 ) ( P3 ) 1.0 ;
end { def groups
```

will produce exactly the same group structure as the 15 lines of SYM file code above, except that it will name the groups C1, E1 and CE-1, rather than G1, G2 and G3, respectively. That is, C1 (the *cause* group) will be connected to P1, P2 and the semantic CAUSE; E1 (the *effect* group) will be connected to P3 and the semantic EFFECT; and CE-1 will be connected to groups C1 and E1, to the semantic CAUSE-EFFECT and to one or more semantic units representing its causal strength. The 1.0 after the second close parenthesis represents the causal strength (on a scale from 0 to 1). It is an optional parameter; if left out, Build will assign a causal strength of 1.0 by default. LISA represents causal strengths as connections from the top-level group (e.g., CE-1) to *causal strength* (CS) semantic units. For example, a causal strength of 1.0 is represented as a connection of 1.0 to the semantic unit CS_1.0 ("causal strength 1.0"); a strength of 0.9 would be represented as a connection to CS_0.9; and a strength of 0.95 would be a connection of weight 0.5 to CS_0.9 and a connection of weight 0.5 to CS_1.0 (i.e., causal strength is coarsely coded on the semantic units). Note that this scale does not currently allow negative (i.e., preventive) causal powers. I hope to add this ability in the near future. Note that Build automatically names these groups for you (e.g., if you define a second and third causal relations the resulting groups will be named C2, E2, CE-2 and C3, E3 and CE-3, respectively), which is convenient, but it means you need to keep track of the names it generates for use in defining sequences (as discussed below).

Sequence Construction Commands:

After you issue the `Sequence` command, Build expects you to issue either the `Done` command (ending the definition of the simulation sequence) or one of eight high-level sequence definition commands:

`Driver=[ index ]` tells Build to designate Analog[*index*] as the driver. Indices are assigned to analogs in the order in which they are defined. ***Note*** that Python indexes starting at zero, so the first analog you define will be Analog[0], the second will be Analog[1], etc.

`Recip=[ index1 index2… index-n ]` tells Build to designate Analogs *index1*… *index-n* as recipient analogs. Typically there will be 0 to 2 recipient analogs: During analogical retrieval all analogs (except the driver) will be in LTM, so there will be no recipients. During analogical mapping and inference there will be one recipient (the target). And during inference and schema induction there will be two (the recipient and the emerging schema). Although I have never found it necessary to have more than two recipients at a time, the code will allow you to designate as many as you like. Be aware, however, that LISA only learns mapping connections to/from units in the driver, not among recipient analogs. Thus, if Analog 0 is the driver and Analogs 1 and 2 are recipients, LISA will learn mapping connections between 0 and 1, and between 0 and 2, but not between 1 and 2. As of this writing (July 24, 2007) analog

retrieval (described in Hummel & Holyoak, 1997, *Psych. Review*) has not been implemented in Python LISA.

SSL_On tells LISA to license self-supervised learning (i.e., for analogical inference and schema induction) regardless of the state of the mapping between the driver and the recipient(s).

SSL_Off tells LISA to *prohibit* self-supervised learning (i.e., for analogical inference and schema induction) regardless of the state of the mapping between the driver and the recipient(s).

SSL_OK tells LISA to figure out for itself whether to license or prohibit self-supervised learning as a function of each recipient's mapping to the driver.

SIM_On tells LISA to compute the similarity between the driver and the recipient(s). This computed similarity does not affect the model's behavior. It is only a reported "similarity judgment" for the benefit of the modeler.

SIM_Off tells LISA to stop computing similarity. By default, similarity computation is off. Once it is turned on, it remains on until the `SIM_Off` command is encountered. When it is on, similarity is computed each time the mapping connections are updated.

Order=[<*various arguments*>] tells Build to define the order in which propositions fire. Once `Order=[` is issued, there is a variety of ways to tell Build/LISA to set the firing order of propositions. The simplest is just to tell it the exact firing order. For example:

```
Order=[ P1 h P2 h ]
```

tells LISA to fire P1, then update the mapping connections (the `h` is the directive to update the mapping connections), then fire P2, then update the mapping connections again. Any propositions that fire between updates of the mapping connections are in the phase set together and are effectively processed in parallel. For example, the instructions above place P1 and P2 into separate phase sets, processing P1 fully before moving onto P2 (so that the mappings discovered for P1 stand to affect those discovered for P2). By contrast:

```
Order=[ P1 P2 h ]
```

places P1 and P2 into the same phase set, processing them in parallel. If you run LISA using the neurally-plausible working memory option (the default), then there will be a limit on the number of propositions the model can place into the phase set together; beyond that limit, separate SPs will fail to fire in systematic asynchrony. (The exact numerical value of that limit depends on the values of various parameters, especially `DriverInhibition`.)

Telling LISA exactly what to fire when is a kind of "hand-holding" that is often useful in the early stages of simulation (and code) development. However, once you've got your simulation to the point where you can understand how it works, it is often more useful to allow the model to set its own firing order based on the importances of, and support relations among, its propositions. One way to do so is to designate a random firing order. For example:

```
Order=[ R ( 10 1 ) ]
```

tells LISA to fire 10 phase sets in a random order (influenced by their importances and the support relations among the propositions), placing one proposition into each phase set. In general, the syntax:

```
Order=[ R ( n p ) ]
```

tells LISA to fire *n* phase sets in a random order, placing *p* at a time into each one. In between complete hand-holding and completely random firing, it is also possible to tell LISA to do group-based random firing:

```
Order=[ G1 ( n p ) ]
```

tells LISA to fire *n* phase sets composed of propositions belonging to group G1 (chosen in a random order), placing *p* propositions into each phase set. Note that this command is hierarchical. For example, imagine that G3 has no propositions of its own, but instead takes G1 and G2 as members, where G1 has P1 and P2 as members and G2 has P3 as a member. In this case, the command:

```
Order=[ G3 ( n p ) ]
```

tells LISA to fire *n* phase sets chosen from propositions P1…P3 (i.e., the members of G1 and G2, which are members of G3) placing *p* of them into each phase set.

To place fire control into the hands of causal groups defined using the Cause command (as discussed above) you'll need to keep track of the names Build automatically assigns to those groups. (Recall that Build names top-level causal relation groups (i.e., those linking cause groups to effect groups) CE-*i*, where *i* is the index of the group, e.g., CE-1 for the first, CE-2 for the second, etc. Thus, for example,

```
Order=[ CE-1 ( 5 1 ) ]
```

tells LISA to fire 5 phase sets chosen from the propositions belonging to CE-1 (and its member groups, C1 and E1), placing 1 proposition into each phase set.

Of course, these conventions can be combined in any way you like. For example:

```
Order=[ P1 h P2 h G1 ( 3 1 ) R ( 10 1 ) ]
```

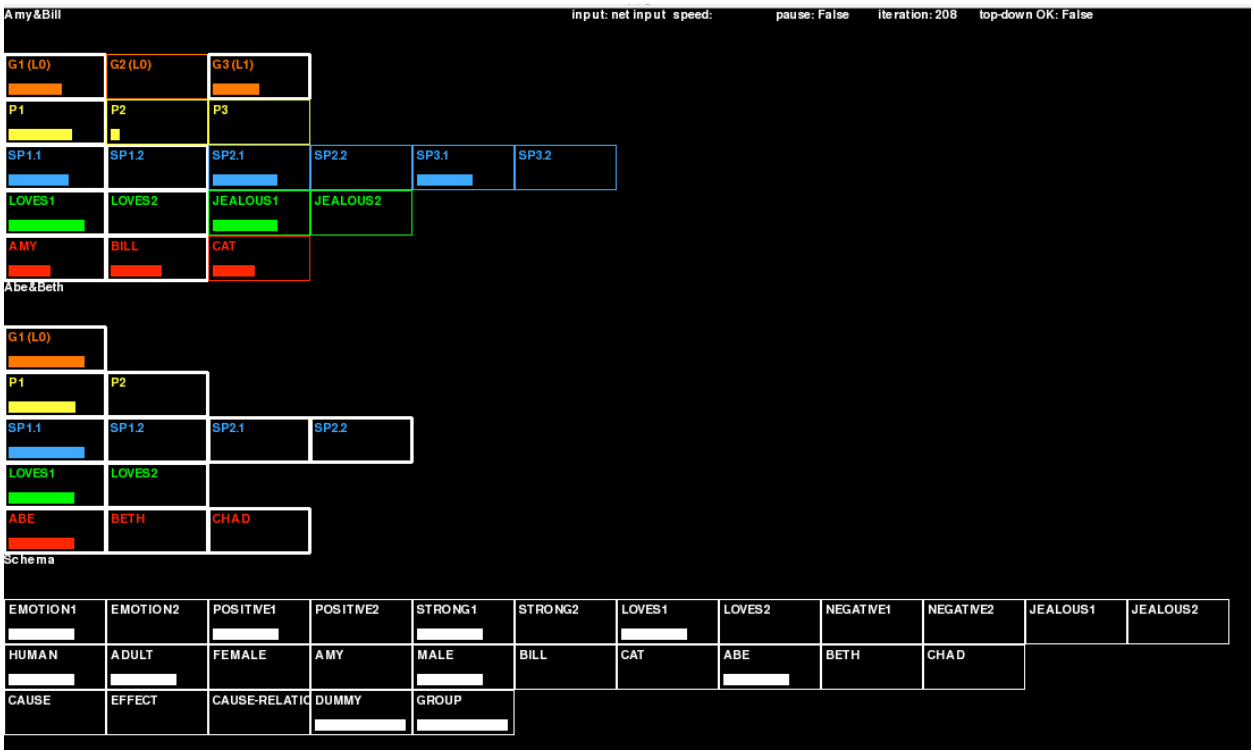
says, "fire P1, then update the mapping connections, then fire P2, update the mapping connections, then fire three props belonging to group G1 (in a random order), placing one proposition into each phase set, then fire 10 props in a random order, placing one at a time into the phase set."

Combining all these ingredients, here is a sample sequence:

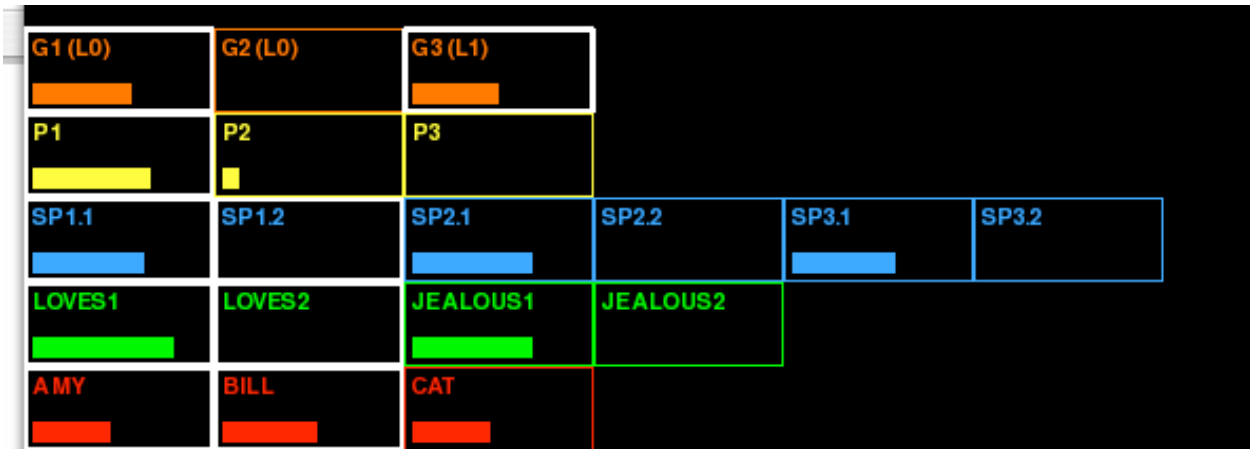
```
Sequence                  { tell Build to define the sequence
   SIM_On                 { turn similarity-computation on
   Driver=[ 1 ]           { make Analog 1 the driver
   Recip=[ 0 ]            { make analog 0 the recipient
   Order=[ P1 P2 h R ( 2 1 ) ]
   { the above first fires P1 and P2 in the phase set together
   { then two props, randomly chosen, each by itself in the phase set
   Driver=[ 0 ]           { make Analog 0 the driver
   Recip=[ 1 2 ]          { make Analogs 1 and 2 the recipients
   SSL_OK                 { make LISA decide when to initiate ssl
   Order=[ G1 ( 3 1 ) ]   { fire three propos from G1 randomly
Done                      { with the sequence and the whole sym file
```

Interpreting the Model's Output

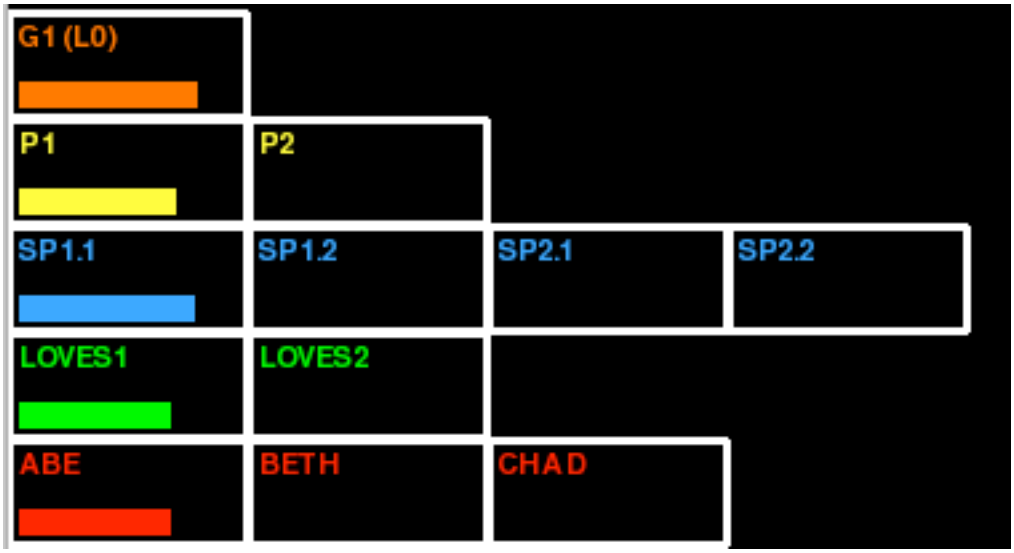
Graphical Output (pygame window):



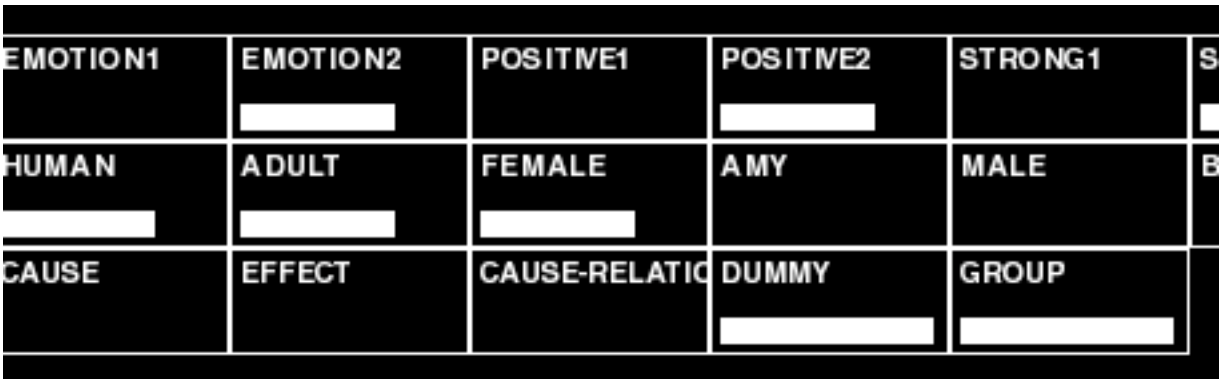
This is a screen shot of the entire pygame window on iteration 208 (i.e., very early in the run) of a run of `lovetri/lovetri7`. Rectangles are units. The top cluster of colored rectangles are the units in "Amy&Bill" (the source analog, and at the time of this screen shot, the recipient). The middle cluster of rectangles is the "Abe&Beth" analog (the target, and in this screen shot, the driver). The bottom cluster of white rectangles are the shared semantic units. This image is pretty low-resolution, so let's zoom in to look at the units more closely…



This image is a close-up of the "Amy&Bill" (source and, currently, recipient) analog. Orange units are group units (G1…G3), yellow units are P (proposition) units, blue are SPs, green are predicate units and red are object units. (Propositions that are currently acting as arguments to other propositions are depicted in red, like objects, rather than yellow.) Each rectangle shows the unit's name. SPs are named for the proposition to which they belong and their place in that proposition. For example, SP1.2 is the second role-binding (the 2 after the period) of the first proposition (the 1 before the period). The colored bar inside each rectangle shows the unit's activation. In the figure above, P1 is highly active (large yellow bar), P2 is slightly active, and P3 is inactive. Rectangles with white frames are units that have been retrieved into working memory; the others are in active memory but not working memory. Only units in working memory can learn mapping connections.



This image shows the "Abe&Beth" (target and driver) analog at the same instant in time. Currently, the "Abe+lover" role of the *loves* (Abe, Beth) proposition (P1) is firing. The activations are "cleaner" in this analog than in the other because this analog is acting as the driver rather than the recipient.



This image zooms in on a few of the semantic units (at a slightly later point in the simulation than the previous images). As with the other kinds of units, the length a unit's the bar is proportional to the unit's activation: "dummy" is highly active (activation approximately 1.0), "female" is slightly less active (activation closer to 0.7 or 0.8) and "male" in inactive.

The Text Output to the Terminal:

At the beginning of each phase set, LISA writes identifying information to the Terminal window:

```
================================
Sequence element 0
    Driver: Abe&Beth
    Recip : Amy&Bill
    Fire control: props
    Phase set   : P1    P2
```

It specifies the index of the sequence element (i.e., the current element in the list of phase sets to run), the name of the driver analog, the name(s) of any recipient(s), "fire control" and the contents of the phase set (the phase set depicted above contains both P1 and P2). "Fire control" describes the basis for deciding which proposition(s) to place in the phase set. "Fire control = props" means that the sym file specifies which propositions to place in the phase set (in the case of the phase set depicted above, the command was `Order=[ P1 P2 h ]`). "Fire control = random" means that props are chosen at random to fire (e.g., the command `Order=[ R ( 1 2 ) ]`). "Fire control = groups" means group-based random firing (e.g., the command `Order=[ G1 ( 1 2 ) ]` or `Order=[ CE-1 ( 1 2 ) ]`, etc.).

At the end of the phase set, LISA writes the current state of the mapping connections to the Terminal window:

```
P1 <-> P1 wt = 0.992
SP1.1 <-> SP1.1 wt = 0.993
LOVES1 <-> LOVES1 wt = 0.982
AMY <-> ABE wt = 0.966
SP1.2 <-> SP1.2 wt = 0.993
LOVES2 <-> LOVES2 wt = 0.982
BILL <-> BETH wt = 0.980
G1 (L0) <-> G1 (L0) wt = 0.982
P2 <-> P2 wt = 0.992
SP2.1 <-> SP2.1 wt = 0.993
SP2.2 <-> SP2.2 wt = 0.993
CAT <-> CHAD wt = 0.958
```

It may also write miscellaneous diagnostic information to the Terminal window. If you wish to make it stop doing so, search for `print` statements near the text `# DIAG` (my convention for reminding myself where I have placed diagnostics in the code) and either delete them or comment them out by preceding them with a pound sign (#). Your changes to the python code will take effect as soon as you save the changed file.

Output (.run and .bat) Files:

Output files can be comparatively large (several pages). The blue Courier text below is from `lovetri/lovetri7.run`: The output file from one run of `lovetri7.sym`. The black Times New Roman text explains the blue text.

```
Note: Whole thing: Inference schema induction, LISA-guided
initiation of SSL
```

The text above is the result of a `Note:` command in the sym file.

The next thing LISA writes to the output file is the set of run parameters used on the run (below).

```
* * * * * * * * * * * * * * * * * * * * *
Parameters values:
    Neurally-plausible WM
    Semantic Noise = 0.0000
    Semantic Death = 0.0000
    Attention = 1.0000
    Within-group support = 1.0000
    Driver Inhibition = 1.0000
    Recip Inhibition = 1.0000
    Hebb Learning Rate = 1.0000
    Bail Upon Settling = False
    Using Vers142 Mapping Algorithm
* * * * * * * * * * * * * * * * * * * * *
```

Next is the sequence of operations on the run. This sequence is the result of the `Sequence` specified in the sym file. (I'm sorry for the small font below. Each line is too long to show in 12-point font.) Each line shows: the sequence index, the driver analog, the recipient analog(s), the value of Fire control, and the contents of the phase set:

```
Sequence:
Seq. 0    Driver:Abe&Beth    Recip : Amy&Bill    Fire control:props    Phase set: P1 P2
Seq. 1    Driver:Abe&Beth    Recip : Amy&Bill    Fire control:props    Phase set: P1
Seq. 2    Driver:Abe&Beth    Recip : Amy&Bill    Fire control:props    Phase set: P2
Seq. 3    Driver:Abe&Beth    Recip : Amy&Bill    Fire control:props    Phase set: P1
Seq. 4    Driver:Abe&Beth    Recip : Amy&Bill    Fire control:props    Phase set: P2
Seq. 5    Driver:Amy&Bill    Recip : Abe&Beth Schema    Fire control:props    Phase set: P1
Seq. 6    Driver:Amy&Bill    Recip : Abe&Beth Schema    Fire control:props    Phase set: P2
Seq. 7    Driver:Amy&Bill    Recip : Abe&Beth Schema    Fire control:props    Phase set: P3
```

Next it saves the mapping connections. A subset of those generated on this run are shown below. They are organized in terms of "From <*analog x*> to <*analog y*>", for all *x* and all *y* (but recall that LISA will not learn connections to/from any *x* except to/from the driver). Within analog pairs, the mapping connections are organized according to type of unit. Only connections with weights greater than zero are saved to file.

```
* * * * * * * * * * * * * * * * *
* * * Mapping Connections * * *
* * * * * * * * * * * * * * * * *

From Amy&Bill to Abe&Beth:
  Groups:
    From G1 (L0) to:    G1 (L0) = 0.960
    From G2 (L0) to:    "G2 (L0)" = 0.970
    From G3 (L1) to:    "G3 (L1)" = 0.879
  Props:
    From P1 to:    P1 = 0.992
```

E.g., The line above says "the connection from P1in Amy&Bill to P1 in Abe&Beth has a weight of 0.992"

```
    From P2 to:    P2 = 0.992
    From P3 to:    "P3" = 0.992
  Preds:
    From LOVES1 to:    LOVES1 = 0.970
    From LOVES2 to:    LOVES2 = 0.970
    From JEALOUS1 to:    "JEALOUS1" = 0.969
    From JEALOUS2 to:    "JEALOUS2" = 0.969
  Objs:
    From AMY to:    ABE = 0.970
    From BILL to:    BETH = 0.970
    From CAT to:    CHAD = 0.970

From Amy&Bill to Schema:
  Groups:
    From G1 (L0) to:    "G1 (L0)" = 0.970
    From G2 (L0) to:    "G2 (L0)" = 0.970
    From G3 (L1) to:    "G3 (L1)" = 0.879
  Props:
    From P1 to:    "P1" = 0.992
    From P2 to:    "P2" = 0.992
    From P3 to:    "P3" = 0.992
  Preds:
    From LOVES1 to:    "LOVES1" = 0.970
    From LOVES2 to:    "LOVES2" = 0.970
    From JEALOUS1 to:    "JEALOUS1" = 0.969
    From JEALOUS2 to:    "JEALOUS2" = 0.969
  Objs:
```

```
    From AMY to:     "AMY" = 0.970
    From BILL to:     "BILL" = 0.970
    From CAT to:     "CAT" = 0.969

From Abe&Beth to Amy&Bill:
  Groups:
    From G1 (L0) to:     G1 (L0) = 0.960
    From "G3 (L1)" to:     G3 (L1) = 0.879
    From "G2 (L0)" to:     G2 (L0) = 0.970
  Props:
    From P1 to:     P1 = 0.992
    From P2 to:     P2 = 0.992
    From "P3" to:     P3 = 0.992
  Preds:
    From LOVES1 to:     LOVES1 = 0.970
    From LOVES2 to:     LOVES2 = 0.970
    From "JEALOUS1" to:     JEALOUS1 = 0.969
    From "JEALOUS2" to:     JEALOUS2 = 0.969
  Objs:
    From ABE to:     AMY = 0.970
    From BETH to:     BILL = 0.970
    From CHAD to:     CAT = 0.970
```

Finally it shows any units inferred by any analogs. Inferred units take the name of the driver unit that caused them to be inferred, but put that name inside quotes. Hence, an object such as "Bill" does not really represent Bill, literally, but instead represents whatever structure was inferred by the self-supervised learning algorithm in response to Bill in the driver.

```
* * * * * * * * * * * * * * * * *
* * * * Inferred Units  * * * *
* * * * * * * * * * * * * * * * *

* * * * * Analog Abe&Beth * * * * *

Groups:
"G3 (L1)":  "G2 (L0)" CAUSE (0.723) CAUSE-RELATION (0.717)
"G2 (L0)":  "P3" EFFECT (0.723) CAUSE-RELATION (0.717)
```

Abe&Beth inferred two groups, shown above: "G3" is connected to group "G2" and the semantic features "cause" and "cause-relation"; "G2" is connected to "P3" and the semantics "effect" and "cause-relation". These inferences are incorrect and demonstrate that groups did not yet work correctly in the version of LISA that generated this output. ("G3" should be connected to both "G2" and G1.) These routines *do* work correctly in version Beta 3, so you will not see these errors when you run Beta 3.

```
Props:
"P3":  [ "JEALOUS1" + ABE ] [ "JEALOUS2" + CHAD ]
```

Abe&Beth inferred one proposition: *jealous* (Abe, Chad). The format above shows the prop's SPs as brackets containing the predicate (role) and argument to which they are connected.

The semantic content of the inferred relation *jealous* (predicate units "JEALOUS1" and "JEALOUS2") is listed below. Semantic units are listed by name with the connection weight to the predicate (or object) unit in parentheses. Semantic unit names are left-justified in proportion to the strength of the connection weight.

```
Preds:
"JEALOUS1":
   EMOTION1(1.000)
           POSITIVE1(0.171)
   STRONG1(1.000)
           LOVES1(0.171)
   NEGATIVE1(0.962)
   JEALOUS1(0.962)

"JEALOUS2":
   EMOTION2(1.000)
           POSITIVE2(0.171)
   STRONG2(1.000)
           LOVES2(0.171)
   NEGATIVE2(0.962)
   JEALOUS2(0.962)


Objs:
```

The inferences made by the schema are shown below. Its entire structure is inferred: Three groups, three propositions, two relations (four predicate units) and three objects. Note how the inferred objects "AMY", "BILL" and "CAT" are strongly connected to the semantic features "HUMAN" and "ADULT" and only weakly connected to more specific semantics; that is, each refers to a "generic person". The inferred propositions below can thus be interpreted as stating:

P1 *loves* (person1, person2)
P2 *loves* (person2, person3)
P3 *jealous* (person1, person3)

The inferred groups can be interpreted as stating that P1 and P2 jointly cause P3. (This interpretation is a bit generous in the case of the output shown below due to the errors noted above. Version Beta 3 does not make these errors.)

```
* * * * * Analog Schema * * * * *

Groups:
"G3 (L1)":  "G1 (L0)" "G2 (L0)" CAUSE (0.723) CAUSE-RELATION
(0.717)
"G1 (L0)":  "P1" "P2" "P3" CAUSE (0.723) CAUSE-RELATION (0.717)
"G2 (L0)":  "P3" EFFECT (0.723) CAUSE-RELATION (0.717)
```

```
Props:
"P1":  [ "LOVES1" + "AMY" ] [ "LOVES2" + "BILL" ]
"P2":  [ "LOVES2" + "CAT" ] [ "LOVES1" + "BILL" ]
"P3":  [ "JEALOUS1" + "AMY" ] [ "JEALOUS2" + "CAT" ]

Preds:
"LOVES1":
   EMOTION1(1.000)
   POSITIVE1(1.000)
   STRONG1(1.000)
   LOVES1(1.000)

"LOVES2":
   EMOTION2(1.000)
   POSITIVE2(1.000)
   STRONG2(1.000)
   LOVES2(1.000)

"JEALOUS1":
   EMOTION1(1.000)
           POSITIVE1(0.171)
   STRONG1(1.000)
           LOVES1(0.171)
   NEGATIVE1(0.962)
   JEALOUS1(0.962)

"JEALOUS2":
   EMOTION2(1.000)
           POSITIVE2(0.171)
   STRONG2(1.000)
           LOVES2(0.171)
   NEGATIVE2(0.962)
   JEALOUS2(0.962)

Objs:
"AMY":
   HUMAN(1.000)
   ADULT(1.000)
         FEMALE(0.423)
         AMY(0.423)
       MALE(0.595)
       ABE(0.595)
"BILL":
   HUMAN(1.000)
   ADULT(1.000)
       FEMALE(0.595)
         MALE(0.423)
```

```
          BILL(0.423)
        BETH(0.595)
"CAT":
   HUMAN(1.000)
   ADULT(1.000)
          FEMALE(0.423)
        MALE(0.595)
          CAT(0.423)
        CHAD(0.595)
```

Parameters

The `(M)odify parameters` option in the Main Menu (`LISA>` prompt in the Terminal window) allows you to modify several parameters that affect the model's behavior. Here I detail what each of these parameters does. After you enter `M` at the `LISA>` prompt, a submenu will ask you whether you wish to modify `(f)ile read/write, (r)un parameters or (d)angerous parameters`.

(F)ile Parameters:

The file read/write parameters tell LISA which data to save to the output file, and whether to run diagnostics during file reading. The first four simply allow you to tell LISA which "Hebbs" (i.e., mapping connections) to save to the output file. The fifth, "`Min Vertical Wt to save`", specifies the threshold for saving an inferred semantic-to-predicate (or -object) weights to save to file; below this threshold, inferred semantics are not saved to file. Finally, "`File Read Diagnostics`" specifies whether LISA will run diagnostics (basically, telling you what Build is doing) as it reads a sym file.

(Ordinary) (R)un Parameters:

These are parameters that you can modify to simulate aspects of cognitive development, cognitive aging, brain damage and other fun stuff (see Viskontas, Morrison, Holyoak, Hummel, & Knowlton, 2004; Morrison, Krawczyk, Holyoak, Hummel, Chow, Miller, & Knowlton, 2004; Hummel & Holyoak, 1997). There are also parameters you can toggle to switch between different modes of operation.

1) Neurally-plausible WM capacity vs. Unlimited WM capacity: LISA's algorithm for establishing systematic asynchrony of firing between separate SPs is based on inhibitory connections between SPs: Because SPs inhibit one another, they fire out of synchrony. This neurally-plausible basis for establishing asynchrony of firing has the very desirable property (from a theoretical perspective, i.e., as a model of human cognition) that it is intrinsically capacity-limited: Only a finite number of SPs can be active simultaneously and successfully fire out of synchrony. This capacity limit is very successful as a model of human WM capacity limits. When you choose the Neurally-plausible WM capacity option, it is this algorithm for establishing asynchrony that you are choosing.

At the same time, however, it can be useful to relax these WM capacity limits, for example, to see what the architecture is capable of in principle or to use LISA as an architecture for pure AI. For this purpose the Unlimited WM capacity option is useful. This option abandons the neurally-plausible algorithm for establishing asynchrony of firing in favor of a decidedly *implausible* algorithm: Make a list of SPs and just fire them in order. The resulting algorithm has no capacity limit, is not neurally or cognitively plausible, and has the capacity to make LISA smarter than people are (or at least much better at making and using analogies). My own simulations with this option suggest that the model has terrific promise as a pure AI engine.

2) Semantic Noise: The value of this parameter, multiplied by a random number between 0 and 1, is added to the input to each semantic unit on each iteration. The default value of this parameter is zero. However, with non-zero values, it can be used for symmetry breaking, for testing the robustness of the model's various algorithms to noise, or for simulating the effects of distraction, fatigue or inattention.

3) Semantic Death Rate: The value of this parameter is the probability that any given connection from a semantic unit to an object or predicate unit will be randomly set to zero at the beginning of a run (i.e., simulating the "death" of the semantic connection). The default value of this parameter is zero. This parameter can be used very effectively to simulate the effects of temporal variant fronto-temporal degeneration (i.e., temporal brain degeneration or damage; see Morrison et al., 2004).

4) Attention: This parameter determines the degree to which a proposition's importance and support from other propositions influence the likelihood of its being chosen to fire when firing is either random (command `Order=[ R ( n p ) ]` under Sequence in the sym file) or group-based (command `Order=[ Gi ( n p ) ]`). Its default value is 1.0. With values less than 1.0, it can be used to simulate the effects of normal aging on analogical reasoning performance (see Viskontas et al., 2004). With values greater than one? We haven't yet tried it.

5) Driver Inhibition: This parameter modulates the ability of SPs in the driver to inhibit one another and thus to establish asynchrony of firing. It only has an effect under Neurally-plausible WM (parameter 1 above). Its default value is 1.0. With values less than 1, it may be useful for simulating aspects of cognitive development (Hummel & Holyoak, 1997), brain damage, inattention, stress and other factors known to influence frontal lobe function.

6) Recipient Inhibition: This parameter modulates the ability of units in recipient analogs to inhibit one another and thus to establish clean mappings to driver units. Its default value is 1.0. With values less than 1, it may be useful for simulating aspects of cognitive development (Hummel & Holyoak, 1997), brain damage, inattention, stress and other factors known to influence frontal lobe function.

7) Hebb Learning Rate: This parameter modulates the rate at which mapping connections ("Hebb connections" in the code) learn weights in response to their buffers. Its default

value in the current code is 1.0. Historically (e.g., Hummel & Holyoak, 1997, 2003) its default value has been 0.9.

8) Bail when recip settles: This parameter only works under Unlimited WM capacity (parameter 1 above). When it is false, each SP in the driver runs for a fixed number of iterations before moving on to the next SP (parameter (13) Phase duration, under Dangerous parameters, below). When it is true, the driver moves on from the current SP to the next SP 10 iterations after the units in the recipient have settled (i.e., their activations have stopped changing). This parameter can be used to derive response-time estimates from LISA and also to increase (slightly) the speed with which the model operates by trimming iterations off the time it takes to run each SP. In future versions of the model, it will hopefully be an option to set this parameter to True under both Unlimited and Neurally-plausible WM capacity.

9) H&H 97/03 Mapping Algorithm vs. Hummel & Green (Vers142) Mapping Algorithm: The LISA model published by Hummel and Holyoak (1997, 2003) uses a very simple algorithm for converting mapping connection buffers into connection weights. Long about 2003 or 2004, John Hummel and (then) grad student, Collin Green (now at NASA-Ames in Northern CA), developed a much more sophisticated (albeit much more complex) algorithm based on Hummel & Holyoak's (1997, 2003) idea that mapping connections are not implemented neurally as connections (i.e., synapses) per se, but rather as neurons in frontal cortex with rapidly-modifiable response properties. (Unfortunately, we have not yet published this new algorithm. But we're working on it. Slowly.) Choosing the H&H9703 value of this parameter causes the model to use the old version of the mapping algorithm; choosing the H&GVers142 value of this parameter (the default value) causes it to use the new algorithm.

10) SSL Threshold: This parameter (default value = 0.7) determines the proportion of a target analog that must map to the source before LISA will license self-supervised learning. This parameter can only affect the model's behavior when the `SSL_OK` command is issued in the `Sequence` section of the sym file.

11) Within-group support: By default, proposition in the same group in an analog *support* one another (i.e., in order to influence on another's probably of being chosen randomly to fire; see Hummel & Holyoak, 1997, 2003). This parameter determines the strength of that support. Its default value is 1.0.

50) Hummel & Holyoak 03 Parameter Suite: When chosen, this option sets the model's parameters to mimic as closely as possible the parameter values used in Hummel & Holyoak's (2003) version of the model.

51) Default Parameter Suite: When chosen, this option sets the model's parameters to the current default settings.

(D)angerous Parameters:

These are parameters that affect the most basic operation of the model and that can therefore get you into trouble. You might have fun modifying these parameters (e.g., by observing how robust the model is to their values), but doing so is more likely to do harm than good.

1) Prop-to-prop Inhibition: The connection strength with which propositions in a recipient analog inhibit one another. The default value is -1.

2) SP-to-SP Inhibition: The connection strength with which SPs in an analog inhibit one another. The default value is -1.

3) Out-prop: Prop-to-SP: The connection strength with which propositions in a recipient analog inhibit SPs not connected to them. The default value is -1.

4) Out-prop: SP-to-pred: The connection strength with which SPs in a recipient analog inhibit predicate units not connected to themselves. The default value is -1.

5) Out-prop: SP-to-obj: The connection strength with which SPs in a recipient analog inhibit object units not connected to themselves. The default value is -1.

6) Pred-to-SP: The connection strength with which predicate units in a recipient analog excite SP units to which they are connected. The default value is 1.

7) Obj-to-SP: The connection strength with which object units in a recipient analog excite SP units to which they are connected. The default value is 1.

8) Semantic-to-pred: A global (pan-unit) weighting term on the input from semantic units to predicate units in recipient analogs. The default value is 1.5.

9) Semantic-to-obj: A global (pan-unit) weighting term on the input from semantic units to object units in recipient analogs. The default value is 0.5.

10) Retrieval Threshold: The activation a proposition or group unit in a recipient analog must achieve in order to be retrieved from active memory into WM and thus to have the opportunity to learn mapping connections to units in the driver. The default value is 0.4. There are times, especially when the correct mappings are not well-supported by units' semantic overlap (e.g., the boys&dogs simulations), when it is useful to lower this threshold (e.g., to 0.3)

11) Hebb Bias: A global (pan-unit) weighting term on the effect of input arriving via mapping connections on units in recipient analogs. The default value is 2.

12) Iterations per SP (normal WM only): This parameter, which only has an effect under Neurally plausible WM capacity (parameter (1) under Ordinary Run Parameters, above) is used to set the duration of a phase set (in iterations) based on the number of driver SPs

in tat phase set (*Duration = Iterations_Per_SP * Number_of_SPs*). The default value is 330. This value is chosen to allow each SP to fire three times for approximately 110 iterations each time.

13) Phase duration (unlimited WM only): This parameter, which only has an effect under Unlimited WM capacity (parameter (1) under Ordinary Run Parameters, above) is used to set the duration of each phase of a phase set (in iterations). The default value is 75. Each Sp in the phase set fires three times for *Phase_duration* iterations each time.

Components of the Code (.py files)

Version 1.00 of the Python implementation of LISA consists of 9 separate files, listed alphabetically:

build.py contains the simulation parser: These routines read a sym file and use it to construct a network and a simulation sequence.
dataTypes.py defines the key data types and sets the default values of all the runtime parameters.
graphics.py contains the graphical routines.
hebbs.py constains the routines that update mapping connections ("Hebbs" in the code, for their essentially Hebbian learning algorithm).
LISA.py is the main file that coordinates the activities of all the other files. It is lisa.py that one invokes in the Terminal window in order to run the model (`% python lisa.py`).
outFile.py contains the routines for saving simulations results to output (`.run` and `.bat`) files.
parameters.py contains the routines that allow you to modify the runtime parameters from the Terminal window. Note that it does *not* define the default values of these parameters; these values are defined in dataTypes.py.
runLISA.py contains the routines that update the state of the network on an iteration-by-iteration basis
ssLearn.py contains the routines that implement self-supervised learning.

Version Notes and Miscellany

Version 1.00 was last updated July 25, 2007. I think this version of the code is pretty bug-free. (But note the weasel-words.)

The model is complete to the standards of Hummel & Holyoak (2003) (indeed, beyond it in many respects). But since it does not yet implement analog retrieval, it lacks some functionality of the 1997 version of the model.

If you encounter any bugs, or otherwise wish to contact me about the code and/or the model, I would greatly appreciate your feedback. I can be reached at jehummel@uiuc.edu.