Sep 17/15

Preamble
````````

This 'how-to' file is intended to enable developers to work on building/extending the knowledge (schemas, features, and choice trees) of Lissa or variants of it.

The general strategy in LISSA, in stepping through a schema instance, is to associate two major items with each question (or potentially other output) by LISSA: (1) a set of choice trees (set of choice packets) for extracting "gist clauses" -- as a preliminary form of interpretation -- from the user's response, and (2) a set of choice trees (choice packets) for reacting to the user's response.

Both types of knowledge are stored in a single file, such as "rules-for-major-input.lisp", or "rules-for-favorite-class-input.lisp", where we use systematic names of type "rules-for-... ...-input.lisp" that hint at the LISSA question (etc.) that is being responded to.

Under item (1) we will normally have 4 packets, one for "specific" answers, i.e., answers that seem to directly and appropriately respond to LISSA's question (etc.); one for detecting "thematic" answers, i.e., thematically repetitive user inputs; "unbidden" answers that seem to supply extra information, prompted by LISSA's question, that may preempt later LISSA questions; and one for questions (such as reciprocal questions) prompted by LISSA's question (etc.). Note that multiple gist clauses may be extracted, not only because we are applying multiple packest but also because long user responses are broken into 10-word chunks, any one of which may yield gist clauses. However, in general we just use the first gist clause extracted, or perhaps a gist clause for a thematic answer.

Note that gist clauses should be formulated as complete, simple sentences, whose form is designed to be easily used for formulating a reaction. Moreover, they should be "self-contained" (have a clear meaning in isolation), so that the correct reaction packet can be chosen based on the gist clause alone -- it is possible that the appropriate reaction is actually specified in a packet in some other file, rather than the one for the response to the specific LISSA question that the rules-file under consideration. After all, user responses can "stray" from the opic of LISSA's question.
````````

Under item (2) we normally have just one choice packet, which chooses a reaction (if any) based on the gist clauses extracted by the choice packets under item (1). Thus they typically use patterns that are satisfied by the gist clauses.

Moving forward

Immediate progress on Lissa is largely a matter of building choice-tree files analogous to "rules-for-major-input.lisp" or "rules-for-favorite-class-input.lisp" (which are themselves incomplete, esp. the latter) for each of the questions that Lissa asks of the user in the main dialog schema, *lissa-schema*, defined in file "lissa5-schema.lisp". Mind you, Lissa can run without these choice trees -- Lissa will simply move on to her next output after reading the user's input, without injecting any specific reactions to that input, and without avoiding scheduled questions that have already been "accidentally" answered by the user.

However, we also want to code the second dialog with Lissa, and that eventually requires a new schema (structurally very much like *lissa-schema*). More generally, we want to be able to create new types of dialogs, so this discussion here begins with an explanation of schema syntax and meaning (the latter is actually pretty self-explanatory), followed by a discussion of choice packets/trees, and then more specifically how to continue building up Lissa's repertoire of reactions.

1. SCHEMAS

1.1 Schema syntax and meaning

Schemas -- specifically event schemas -- are used to specify dialog and subdialog structures. We need at least one schema to build a Lissa-like system. A simple example of an event-schema definition is

```
(defparameter *simple-schema*

'(Event-schema (((set-of me you) greet-and-run.v) ** ?e)
...
:actions
?a1. (me say-to.v you '(Hi\, I am Lissa. What is your first name?))
?a2. (You reply-to.v ?a1.)
?a3. (me react-to.v ?a2.) ; this might lead to "Nice to meet you, ..."
?a4. (me say-to.v you '(But I\'m afraid I can't hang out -- so\,
bye-bye for now!))
```

))

Here,
- '*simple-schema*' becomes a global parameter (accessible to all the Lisp code); it is conventional to begin and end global parameters with an asterisk, '*'.

- The quote in '(Event-schema ...) is needed to tell Lisp that the value of *simple-schema* is the explicitly given list structure, not a function plus arguments to be evaluated.

- 'Event-schema' is just a mnemonic type indicator that has no current computational significance; by the way, it could also be written as 'event-schema', 'EVENT-SCHEMA', EvEnT-ScHeMa, etc. -- Lisp is case-insensitive, except when we are placing characters in double quotes (e.g., "Lissa") -- but this is not considered an (atomic) word, but rather a character string. Also a character prefixed with escape character '\' (e.g., L\i\s\s\a) retains its case, and multiple characters enclosed in vertical bars, |...| also retain their case (e.g., |Lissa|). Note: L\i\s\s\a and |Lissa| are (atomic) words (indeed the *same* word), not character strings. Anyway, you can avoid these subtleties by not using escape characters -- except as specified below.

- (((set of me you) greet-and-run.v) ** ?e) in the schema is called the "schema header". It is viewed as a logical formula; the predicate is 'greet-and-run.v' (the extension '.v' indicates that this is a verb-like predicate) , and the only argument (the subject) is '(set-of me you)', designating the set consisting of the computer agent and the human dialog partner. (Our logical notation uses English-like infix form, i.e., the subject argument precedes the predicate, and any others follow it.)

The meaning of the header is that if this schema is instantiated and successfully executed, then an event of type
((set-of me you) greet-and-run.v)
takes place. The "wrapper" (... ** ?e) provides a variable ?e referring to this overall dialog event.

However, schema headers are mostly there to remind ourselves what sort of dialog this is -- the code doesn't actually care about the header -- except for any variables occurring in it, *excluding* the event variable in (... ** ?e); specifically, variables in the header are used in the code to pass arguments, if any, to the schema; no need to worry about that yet...

- The dots '...' in the example schema just indicate that certain kinds of information can be placed in a schema ahead of the actions,

(such as preconditions, and constraints on participating entities)
but these are can be omitted for our present purposes.

- The ':actions' keyword indicates that the rest of the schema consists
of alternating action variables and formulas, where the formulas
(predications) specify particular actions. (If another keyword
appears before the end is reached, this will signal the end of
the action specifications and the beginning of other data such
as goals and temporal constraints, but again this need not
concern us here.)

The variables must start with '?' and end with '.', i.e., the period
is actually part of the variable. Conventionally, we use '?a1.', ?a2.',
etc. for action variables. (More accurately, these are *propositional*
variables, standing for the proposition that the specified action
occurs; furthermore, by convention, when we detach the final '.' from
such a variable, we get an event variable, denoting the event of the
action occurring. Once more this need not concern you, but if you're
interested in the exact meaning of an action specification
?ai. wff
where wff is a formula, it is (?ai. = (that (wff ** ?ai))); i.e.,
'?ai.' stands for the proposition that the event '?ai' characterized
by formula wff occurs.)

- The actions are assumed to occur in the order given. (More generally,
we can specify temporal constraints on actions, but as noted that's
not needed here.)

- One point to be careful about in 'say-to.v' actions is that commas,
apostrophes (quotes), colons, and semicolons should be preceded by
escape character '\', to avoid tripping up Lisp. For example, write

(me say-to.v you '(Let\'s talk about it\; you first\, and then me!))
rather than
(me say-to.v you '(Let's talk about it; you first, and then me!)).

Exclamation marks and question marks are unproblematic, and periods
are also "safe" as long as they don't occur in isolation or at the
beginning of a word (so using '...' or '.mailrc' in an output are
no-no's). Of course, one could liberalize all this by making the
schema-loading code responsible for dealing with these trivia, but
right now, schema-loading is treated as a simple assignment of the
schema, as a Lisp list structure, to the name of the schema.

- The three types of actions illustrated by '*simple-schema*' are
all we need for now, though the code also provides in principle
for the following additional types of dialog actions by the system:
(me describe-to.v you <entity>)

(me suggest-to.v you <action-proposition>)
(me ask.v you <nominalized query>)
(me say-hello-to.v you)
(me say-bye-to.v you)
where, e.g.,
<entity> = Garbage-Plate.name;
<action-proposition> = (that (you provide-to.v me
(K ((attr extended.a)
(plur answer.n)))))
(perhaps realizable as (me say-to.v you '(Tell me more)))
<nominalized query> = (ans-to (wh ?x (you have-as.v major.n ?x)))
(perhaps realizable as (me say-to.v you '(What is your major?))).
These options lay the foundation for later actual language *generation* from symbolic knowledge and intentions, but need not concern us at this point.

- When there are no more actions, the dialog is over. However, if this is a *subdialog* that can be used to expand a step in another, larger dialog schema, then the dialog may well continue.

1.2 Comments on how the plan manager works
``````````````````````````````````````````

The following further comments are for understanding only (e.g., in error tracing), but are not needed for schema or choice tree design.

Schemas are not used directly, but rather the plan manager starts by making a copy of a schema (which now gets the type indicator 'plan', rather than 'event-schema'), and partially instantiating it. Partial instantiation means that (a) variables in the schema (now plan) header are replaced by whatever values have been provided via choice trees, throughout the copy of the schema; (b) the action variable (e.g., '?a1.') immediately following the ':action' keyword is replaced by a freshly generated action constant such as 'EP0752.' or A285035.', which is substituted for the action variable throughout the copy of the schema. Furthermore, the new action constant without the final '.' (e.g., 'EP0752') is substituted for for all occurrences (if any) of the episodic variable (in the example, '?a1') derived from the action variable by omission of the '.'.

When further steps of a plan (that started as a schema copy) have been executed, the remaining steps (if any) will still consist of alternating action variables and action formulas. In preparation for performing the next action, much the same substitutions are made as in the initialization of a plan; in particular, an action proposition name (with a final period) and corresponding episode name (without the final period) are substituted for the first remaining action variable and corresponding episode variable through-
``````````````````````````````````````````

out the remainder of the plan. Then the action is implemented, either directly (if it is a (me say-to.v you ...) action), or via a named 1-step or multi-step subplan, created "on the fly" using choice trees and possibly additional schemas. Any step can be either primitive (immediately executable) or nonprimitive (requiring elaboration into a subplan). Currently the only primitive steps are of type (me say-to.v you '(...)), or (you paraphrase.v '(...)), where '...' is an explicit word sequence.

The former type of primitive step is either present from the outset in a schema, or is generated by implementation a (nonprimitve) step of type (me react-to.v <name of a user input>) -- see next subsection.

The second type of primitive step, using the predicate 'paraphrase.v', is generated internally by the system when implementing an action of type (you reply-to.v <name of a system output>). I.e., "implementing" a user reply consists of (i) reading an input, (ii) deriving one or more "gist clauses" from it that try to make the content of what the user said explicit and context-independent (using the question that prompted the user's reply as context), and (iii) generating a subplan containing one or more 'paraphrase.v' steps, representing the system's understanding of what the user said. The idea behind using the 'paraphrase.v' predicate, with a gist clause as argument, is that the system treats the actual input from the user (often condensed and ambiguous when considered out of context) as a paraphrase of a wgist clause the user intended to communicate. Processing of a 'paraphrase.v' action consists of nothing more than attaching the gist clause it specifies as a property of the action name, so that the system can then react to that gist clause as if the user had spoken it directly.

1.3 The idea behind "reactions"

As already noted, the '(me react-to.v ...') formulas may lead to specific outputs derived via choice trees from user inputs; but these reactions may be empty -- if the plan manager can't find anything relevant to say, it will move on to the next step in the schema.

If you read the previous subsection, you will already have an idea of what "gist clauses" are. Lissa's choices of reactions are not based directly on its own verbatim outputs and the user's verbatim inputs but on "gist clauses" derived from those Lissa outputs and user inputs. The gist clauses for Lissa outputs are expected to be supplied directly in whatever file specifies the schema containing those outputs. For example, if you look in file "lissa5-schema.lisp" you will find a 'mapcar' command that associates one or more gist clauses with many Lissa outputs (as named by action variables in the schema), primarily the ones that contain a question that the user

is expected to reply to (which means that we want to be able to make sense of these replies in the context of Lissa's question). For example, for Lissa's output "I am a senior computer science major\, how about you?", the gist clauses stored are
((I am a computer science major \.) (What is your major ?))
Note the explicit form of the question, which will provide the context for making sense of the user's reply. For example, a brief reply like "physics" can mean many things in different contexts, but in the context of the question "What is your major?", it clearly means "My major is physics." Lissa's actual question, "how about you?" would not have provided an adequate context for disambiguation of the user's reply!

In the same way, reactions (if any) by the system to users' replies need the context of what the replies were in an explicit form, to be able to react appropriately; i.e., we need gist clauses derived from the user replies. For example, the reply "physics" to the question about the major can only be reacted to in a context-independent way once we expand it to a gist clause like "My major is physics". So this enables a reaction by the system such as "It looks like we\'re both scientifically minded!". (This reaction would not be appropriate if the reply "physics" had been in answer to the question, "Do you dislike any subjects in your major?").

We note here that gist clauses are intended as a sort of "poor man's semantics" in this stage of our work. Ultimately, we would like to automatically derive the system's outputs by generative mechanisms that map (unambiguous) internal formulas to natural-sounding output, rather than providing all the output texts explicitly. Further, we want to derive logical interpretations of user inputs by using SR, semantic parsing and context-dependent disambiguation mechanisms,
rather than choice trees; however, we're postponing this -- and it will require knowledge bases consisting of hierarchies of verbal interaction schemas, where these are probably deployed using hierarchical matching not unlike the traversal of choice trees.

As a gesture in the direction of future work, "lissa5-schema.lisp" currently contains a 'mapcar' that supplies logical interpretations for some Lissa outputs (where we imagine the actual outputs to be generated from these formulas), especially for questions whose answer may already have been supplied earlier, so that the question should be bypassed if it can be established inferentially that the answer to the question is already known. These formulas are not used currently -- but looking at them can give you some idea of the (quite English-like) Episodic Logic knowledge representation, which is handled by the EPILOG inference engine.

2. CHOICE PACKETS & TREES

We're ready to consider the design of various types of choice trees. A word about terminology: In the various files, both "choice packets" and "choice trees" are frequently occurring terms. The distinction is just that a choice packet is the specification, in a rules file, of a choice tree to be constructed from the packet. The term "packet" is appropriate insofar as the rules you specify in a packet are just provided as a list, with a "depth index" prefacing each rule. The function 'READRULES' forms a choice tree from a choice packet, making the root of the tree whatever atom is supplied as first argument of 'READRULES'; (the second argument is the list of rules and their depth indices). It's also fairly natural to talk of the choice trees as "decision trees" (a standard AI concept, particularly in machine learning). But decision trees typically just decide among a fixed set of class labels, using a tree of features that are checked against an input in a sequential root-to-leaf traversal. By contrast, our choice trees can "bottom out" in arbitrary outputs, in particular ones that contain some pieces gleaned from the final (most specific) match against the input. So choice trees are actually input-to-output *transducers*, not mere classifiers (though they can certainly be used as such).

2.1 Choice packet syntax

Rules specified in a choice packet are either match rules or output rules. A match rule is specified as a single list of words and digits, e.g., (0 you find 3 hard 0), which will match any word list containing "you find" and, at most 3 words later, "hard";

An output rule is specified as an output list or atom, followed immediately by a two-element list containing a latency number and a "directive" -- a keyword indicating the kind of output being specified.
e.g., (No\, you don\'t find 4 especially hard.) (0 :out);
this can supply the explicit output "No\, I don\'t find computer science especially hard.", assuming that the fourth item in the match rule above, namely the digit 3, was matched by "computer science", and the match rule is positioned in the packet just "above" the output rule (more details about packet structure below). The conversion from "you" to "I" in the actual output is due to the fact that the system uniformly takes the user's perspective in its rules; i.e., in both the match rule and the output rule above, "you" is the system (so we are considering a question from the user addressed to the system); so in the system's reply, "you" is changed to "I", and similarly "your"

becomes "my", "I" becomes "you", "my" becomes "your",
etc. -- just something to remember in writing rules!

By convention (for legibility), we put each rule in a choice packet
on a new line. The rule should begin with a depth (level) index,
using index 1 for top-level rules, index 2 for the next level down,
and so on. Furthermore, whenever we go down one level, we indent
the specification of the rule (starting with the depth index) by
one character. For example, the above two rules might be specified
in a choice packet as (assuming that the match rule is top-level)

```
(1 ...
...
1 (0 you find 3 hard 0)
2 (No\, you don\'t find 4 especially hard.) (0 :out)
...
)
```

Requiring this formatting may seem petty, but perspicuity is actually
crucially important in packet design -- and indeed in the design of
any AI system that one intends to be extensible and understandable
by other people! For longer match rules or output rules, don't create
extra-long lines; rather restrict lines to about 70 characters (maybe
occasionally up to 75 or 80 characters), including blanks. Indent
consistently in breaking up a long list; Lisp doesn't care about blanks
and line breaks. Again this is for perspicuity -- the material should
fit comfortably into any reasonable window (or printed page) without
"spill-over".

"Sibling" rules, i.e., ones occurring successively at the same
depth index without an intervening return to a higher depth index,
are alternative ways of matching a word list, or (for output rules)
alternative outputs. For example, suppose that we have the following
pattern of indices occurring in a choice packet:

```
(1 <something>
2 <something>
3 <something> <something>
3 <something> <something>
2 <something> <something>
2 <something>
3 <something>
4 <something> <something>
1 <something>
2 <something> <something>
)
```

Here the two level-1 rules are siblings, and both are match rules
(as is apparent from the fact that they are accompanied by one
item -- the match rule -- rather than two -- an output plus a

(latency directive) pair). Thus they are alternative ways to match a word list). Among the level-2 rules, all but the last are siblings, and thus alternatives. However, the first level-2 rule is a match rule, the second is an output rule, and the third is again a match rule. The two adjacent level-3 rules are two alternative output rules; it's easy to see what kinds of rules the remaining rules are.

So which alternatives are used? For match rules, the answer is whichever rules succeed first in matching the given word list (imagine going top to bottom in the above listing). For output rules, the answer is whichever rule is reached first in a sequence of successful matches leading to that output rule, and is not blocked by its latency requirement. Blocking by latency means that the output rule was used too recently, and we'd rather not use it again just yet, for fear of sounding repetitive. The latency number indicates how many outputs the system should have generated before it uses that same rule again; zero latency means it's never blocked.

Finding an output using a choice tree is essentially a depth-first search that looks for a pattern match to a given word list, and an output corresponding to that match (potentially using matched chunks in the output). When a match rule successfully matches the given word list, we proceed to its first "child" (one level deeper). If this is another (typically, more specific) match rule, we determine if it matches the given word list, and if so, proceed to its first child, if any, or if there is none, "pop up" to the next available higher level, etc. If instead the child of a successfule match rule is an unblocked output rule, its output is constructed (this may involve insertion of material from the previous match, as illustrated earlier); if it is currently blocked, then we proceed to its next sibling, if any; if there are no further siblings, we "pop up" to the next available higher level. For example, suppose we get to the third (second-last) level-2 rule above, in a match attempt; suppose that this rule matches, so that we proceed to its level-3 child; suppose that match fails; then we can't use the output rule at level-4, and so we pop up to the last level-1 match rule; if this succeeds, we'll ouput whatever its level-2 child specifies. If either of these last two rules fails, the result of the choice-tree travesal will be nil.

The internal structure of a choice tree is not simply a listing of rules, but rather an actual tree structure with named nodes (where new node names are automatically generated), one node per rule. All rules have a 'pattern' property specifying a match pattern or an output (which may be a template to be filled in, or fixed). Output-rule nodes have a 'directive' property, a 'latency' property

and a 'time-last-used' property that are directly attached to the rule node. Both match-rule nodes and output-rule nodes have a 'next' property leading to the next sibling (if any) and a 'children' property leading to the sequence of children (if any). But this internal structure need not concern choice packet creators!

2.2 Matching and features

In the simplest cases, our pattern language allows us to look for particular words in a user input (or a gist clause derived from an input), either adjacent to each other or with intervening words that the pattern "doesn't care about". We have already seen that the pattern '(0 you find 3 hard 0)' will match any word list consisting (left-to-right) of 0 or more words, followed by 'you' and 'find', followed by at most 3 words (where punctuation, separated off in preprocessing, also counts as a word), followed by 'hard', followed by 0 or more additional words. Note that such a pattern can detect desired information in a word stream even if that word stream derives from a speech recognizer and as such contains errors. It also allows variations in wording, such as "So, Lissa, do you find computer science hard?", or "Do you find computer science so hard that you have to work all the time?". (This example is actually based on a gist clause associated with Lissa's question "Did you find it hard?", namely "Did you find your favorite class hard?"; by design, gist clauses are more explicit than casual, context-dependent English, but it's not always easy, when you're writing choice packets, to remember the exact wording you may have used in specifying a gist clause. So it's good to allow some variability even in matching against gist clauses.)

When a pattern match succeeds, the word-chunks corresponding to the digits in the match pattern can be accessed by reference to the position of the digits in the match pattern. (This pattern-match method goes all the way back to Joseph Weizenbaum's ELIZA chatbot, circa 1967!). So when we've matched
'(0 you find 3 hard 0)'
against a word list, the digit '1' refers whatever chunk of the input matched the initial '0' in the pattern, '4' refers to whatever matched the '3' in the pattern, and '6' refers to whatever matched the final '0' in the pattern. This is what allows us to construct outputs that "borrow" material from the input. In the earlier output pattern
(No\, you don\'t find 4 especially hard.)
the digit '4' specifies insertion of whatever chunk matched the 4th item in the match pattern, i.e., whatever corresponded to the '3' (up to 3 words) in the match; we previously assumed that the matching words were "computer science", so the result provided by the output

pattern would be (No\, you don\'t find computer science especially hard.) which, as noted, is then converted to its first-person/second-person dual, before being generated by the system.

We should mention two more special symbols that can be used in match rules, namely 'nil' and an initial '-'. An occurrence of 'nil' will match exactly one word (or separate punctuation), and an initial '-' *negates* the pattern, i.e., the pattern matches exactly if the rest of the pattern, without the '-', does *not* match the given word list. For example, the pattern (- 0 computer science 0) matches any word list that does *not* contain the phrase "computer science". An isolated '-' occurring non-initially is treated as ordinary punctuation (dash). (Compounds such as 'African-American or 'hip_hop' are treated as atoms.)
The uses of 'nil' include not only cases where we want to match exactly one word, as in (I\'m very nil 0), where 'nil' will probably match an adjective (as in "I\'m very happy in Rochester"), but also cases where we want to check whether some minimum number of words precede
some other specific word(s) or punctuation. An example in the curr-ently unused) file "choose-doolittle-response.lisp" is the pattern (NIL NIL NIL 0 \. 0), which is used with output rule (1 2 3 4); this can be seen to discard everything after the first period in the user input, *provided* that the material before the period contains at least three words. (In Doolittle, the truncated input is then "cycled back" to Doolittle, as if the user had provided just that input.)

The discussion above actually understates the capabilities of the pattern matching language: We can match not only specific words, but also classes of words, where these have been marked with *features*. For example, the above rule from Doolittle is actually (NIL NIL NIL 0 end-punc 0), where 'end-punc' is a feature that has been assigned to punctuation marks '\.', '!', '?', '\:', '-', and '\;'. Any one feature can be assigned to multiple words using the 'attachfeat' function, e.g.,
(attachfeat '(end-punc \. ! ? \: - \;))
(attachfeat '(index-pron I you me us mine yours ours))
(attachfeat '(ana-pron he she it they him her them hers theirs))
(attachfeat '(wh_ why how what who whose whom when which where))
(attachfeat '(spouse husband wife)
(attachfeat '(friend friends buddy buddies pal pals acquaintance acquaintances friendship friendships))
(attachfeat '(economics economic economy economies microeconomics macroeconomics))
(attachfeat '(challenging-course math physics medicine foreign-language geology life-science economics))
etc.

[We can avoid repeating 'attachfeat' for multiple feature assignments
by using
(mapc '((end-punc \. ! ? \: - \;)
(index-pron i you me us mine yours ours)
...)) .]
Note that features can be freely invented when you are formulating match rules; they can be syntactic categories, synonyms, related words, semantic categories, or whatever else seems to be convenient. Features don't have to be English words, but they should be indicative of what the words tagged with the feature have in common or are related to. Features (whether English words or invented terms) can themselves be assigned arbitrary features, and the higher-level features will "propagate" down to the lower levels. The way the pattern matcher actually works is that it expands each word in a word list to be matched into a list. The list will start with the word, but will also include the features of the word, the features of those features, etc. For example, 'microeconomics' might be expanded into
(microeconomics economics challenging-course academic course).
Thus an occurrence of 'microeconomics' in a given word list can be matched using any member of the above list in a match pattern. We generally try to avoid circularity in the feature hierarchies we build up, but the matcher is designed to be robust in the face of "accidental" circularities -- it will avoid duplicating list elements when expanding words into lists.

What sorts of features are assigned to words is a critical factor in how effectively and broadly we can match user inputs for various purposes, derive gist clauses from them, and hence construct appropriate reactions. One can even envisage marking a large English vocabulary with parts of speech as well as semantic categories, and hence doing sophisticated matching of part-of-speech patterns and semantic patterns, or a mixture of these. Unfortunately, the pattern language does not allow specifying that a word should have multiple, specified features, such as certain lexical, syntactic, and semantic features. To match multiple features to a word, we need to use multiple rules. For example, suppose we wanted to write a pattern for certain kinds of simple sentences that express a positive feeling towards a family member, such as "Of course I love my mother". If we have assigned parts of speech to words, such as 'vbz' for present-tense verbs, 'vb' for untensed verbs, and 'nn' for common nouns, we might first try the pattern
(0 I vbz my nn),
and, at the next level "down" in the choice tree,
(0 I feel-positive-towards my family-member),
assuming that 'feel-positive-towards' is a feature that has been assigned to various verbs (including their tensed inflections!)

such as "love', 'like', 'admire', 'adore', 'respect', etc., and
'family-member' is a feature that has been assigned to 'mother',
'father', 'sister', 'brother', 'grandmother', 'uncle', etc.
(Note, by the way, that output patterns can refer to *any* of the
input chunks that were matched by the preceding match pattern via
position indices, and that includes words that were matched by
a feature. For example, index '3' will refer to whatever word
matched the feature 'feel-positive-towards' in the above pattern,
such as 'love', 'like', etc., which can thus be used in the output.)
A weakness of this strategy is that it may fail to ensure that
the *same* word has the desired multiple features. For example,
the patterns (0 vbz 0) and (0 feel-positive-towards 0) may well
match the same sentence, but different words may supply the matches
to the two specific features, as in "I wish my uncle liked me
more", where 'wish' would match 'vbz' while 'liked' would match
'feel-positive-towards'. (Our more general pattern matching and
transduction language TTT could handle such cases better, but is
a bit more complex to learn and use.)

We should note that the most challenging types of choice packets
that need to be built for a dialog system are those aimed at
"interpreting" the user's inputs in context, which for us currently
means deriving gist clauses from inputs. Once we have those gist
clauses, designing the system's reactions via other choice packets
is relatively straightforward -- because we know from our own
design of the packets that derive gist clauses what the form of
the "interpreted" inputs will take!

2.3 Types of choice tree outputs
````````````````````````````

As has already been explained, output rules in choice packets
are distinguished by the fact that they are immedately followed
(without an intervening depth number) by a 2-element list of form
(latency directive). The directives currently in use are the
following, and have the indicated meaning:

:out {indicating that the rule supplies an explicit ouput
pattern (also called a "reassembly pattern"), possibly
containing digits that are to be filled with chunks from
the input that matched the the preceding match pattern,
at the position in the match pattern indexed by those
digits}

:gist {indicating that the rule supplies either just a reassembly
pattern that constructs a gist clause from the user input,
or a list of two items, where the first is a reassembly
pattern that constructs the gist clause, and the second is
a list of one or more topic indicators; topic indicators
````````````````````````````

are used to link user inputs (converted into gist clauses)
with questions that should be prevented from being asked
by the system, because the gist clause at issue already
answers it, or at least addresses the same topic.}

:subtree {indicating that the rule supplies the name of another
choice tree where the match attempt should continue}

:subtrees {indicating that the rule supplies a list of names of
subtrees, e.g., for extracting different types of gist
clauses from a user input}

:subtree+clause
{indicating that the rule supplies a list of two items,
where the first is the name of another choice tree and
and the second is a reassembly pattern to be used to
construct a clause serving as input in the continued
search (whereas for :subtree the recursion continues
with the original input clause).

:schema {indicating that the rule supplies the name of a schema
to be instantiated, where this schema has no arguments}

:schemas {indicating that the rule supplies a list of names of
schemas, perhaps intended as alternative ways of
carrying on a subdialog}

:schema+args
{indicating that the rule supplies a list of two items:
the name of a schema, and the list of argument patterns
for that schema; these argument patterns may contain
digits indicating that the actual arguments should be
obtained by substituting appropriate chunks from the
preceding pattern match for these digits; this can be
used for example to "feed" multiple gist clauses derived
from a user input to a subdialog schema that is designed
to guide the system's reactions to multiple inputs, such
as a user's reply to a question, where the user provides
a direct answer followed by a question of his/her own.}

At a later point, we might add further types of outputs; e.g.,
patterns that produce a logical formula (an interpretation of the
input) when numeric digits (if any) are filled in; this might use
directive :wff (for "well-formed formula").

3. USING THE ABOVE MACHINERY TO ENHANCE THE LISSA DIALOG(S)

It should already be fairly clear how to construct dialog schemas and choice packets for Lissa-style dialogs, especially given the schemas that already exist, namely *lissa-schema* (in file "lissa5-schema.lisp"), *reactions-to-answer+question* (in file "schema-for-reactions-to-answer-plus-question.lisp") and *reactions-to-question+clause* (in file "schema-for-reactions-to-question+clause.lisp"); and given the choice packets that already exist in files "rules-for-major-input.lisp" and "rules-for-favorite-class-input.lisp". Actually, as far as schemas are concerned, no immediate additions or extensions seem to be needed -- the main schema, *lissa-schema*, is fairly lengthy as it stands, and the two subordinate schemas for handling multiple gist clauses extracted from a reply probably suffice for the time being. However, most of Lissa's questions specified in *lissa-schema* still call for the creation of new rule files supplying choice packets for extracting gist clauses from user replies to Lissa questions, and for choosing reactions to these replies.

Another important file is "choose-gist-clause-trees-for-input.lisp", which defines what is in effect a higher-level choice tree, *gist-clause-trees-for-input*;
this uses Lissa's questions (or other outputs that call for a user response) in gist-clause form (automatically augmented with features) to select a set of choice trees to be used in extracting gist clauses from the user's response. For example, if Lissa's output, in gist clause form, is found to match the pattern
(3 what 4 your major 0),
the choice tree *gist-clause-trees-for-input* returns the list of specialized choice trees
(*specific-answer-from-major-input*
unbidden-answer-from-major-input
thematic-answer-from-major-input
question-from-major-input)
Each of these specialized choice trees will then be used to try to extract gist clauses from the user input. The first of these is aimed at direct answers, such as "My major is history", or "I'm a history major in my junior year", or "I'm doing a double major in history and economics". Outputs from this tree should be gist clauses such as "My major is history", or "My majors are history and economics". (The gist clauses should be simple and straightforward.) The second is aimed at assertions other than direct answers that the user may supply, such as "I'm also doing a minor in computer science". The third is aimed at lengthy replies that may not yield any kind of specific answer or unbidden answer that can be anticipated, but may contain various types of words that suggest a general theme. For example, if multiple mentions of academic subjects are detected in the user's reply, this may indicate that the user is interested in multiple subjects, and thus

it may be useful to create a gist clause expressing this, as a basis for subsequently generating an answer such as "Sounds like you have a broad range of interests". The fourth choice tree is aimed at questions, especially ones at the end of the user's input, since these are the most likely to require a response from Lissa. Often such question are reciprocal; for instance, one user followed his answer concerning his major with the question "What year are you?", having failed to notice that the answer was aleady supplied by Lissa ("I am a senior computer science major").

Similar sets of 4 choice trees are supplied by the higher-level choice tree *gist-clause-trees-for-input* for each of Lissa's questions. Not all of these exist yet, and a major task is to create these sets of choice trees in rule files for each of Lissa's questions. Also, whenever further questions are added to the dialog, corresponding match rules and output rules should be added to *gist-clause-trees-for-input* (after it has been decided what the gist-clause versions of Lissa's new questions should be -- since these are what the patterns in *gist-clause-trees-for-input* should match).

Note that when a user's reply to a Lissa question is more than 15 words long, the system code will apply the first two of the four gist-clause extraction trees (i.e., for specific and unbidden answers) to multiple 10-word segments (chunks) of the reply, overlapping by 5 words. This is necessary since at least in speech recognizer word sequences, and to some extent in typed sentences, sentence boundaries are ambiguous. Gist clauses corresponding to all the successful matches are collected, but currently at most two are used for generating Lissa reactions. (This is why the two subschemas *reactions-to-answer+question* and *reactions-to-question+clause* are probably sufficient fo now.)

We've noted that the most demanding task at this time is creating choice packets for gist clause extraction from user replies (using gist clause versions of Lissa questions as context). However, a good deal of borrowing should be possible for this task from the existing old version-5 files for choosing reactions to user replies. For example, consider the choice tree *reaction-to-hardness-of-class* defined in file "choose-reaction-to-hardness-of-class5.lisp". This starts with the match pattern
(0 neg 0 hard 0),
which looks for an answer such as "I didn't find it particularly hard"; the reaction specified in the choice packet is
(It\'s good I enjoy what comes naturally to me\.)
As noted, the I/you dual of this is intended as actual output.
The match pattern (0 neg 0 hard 0) can be used instead in the packet

defining
specific-answer-from-hardness-of-class-input,
with a corresponding gist clause output such as
(I found my favorite class easy).
This gist clause can subsequently be used in choice tree
reaction-to-hardness-of-class-input
(to be defined in a file "rules-for-hardness-of-class-input.lisp")
to generate a reaction like the one above, i.e., (It\'s good I
enjoy what comes naturally to me\.). So this gets to the reaction
less directly, but with the important advantage that we will have
created a representation of the user's reply (viz., a gist clause)
that can be understood unambiguously outside its immediate context
-- and can be stored in knowledge base *gist-kb*, for possible
future use, e.g., to prevent Lissa from asking a question that is
already answered by a stored gist clause.

Finally, a comment on a weakness of the overall approach that we
should think about, for future revisions. The weakness comes from
the fact that scripted Lissa outputs in a schema don't incorporate
any information previously obtained from the user. For example,
there is currently no obvious way for Lissa to address the user by
his/her name, having learned the name earlier. Similarly, consider
the "hardness of favorite class" exchange we just discussed;
suppose the user previously identified microeconomics as his/her
favorite class, and in reply to the hardness question said "I
didn't find it particularly hard". How can we get Lissa to derive
as a gist clause (I found microeconomics easy), rather than just
(I found my favority class easy)? The problem is that as a context
for interpreting "I didn't find it particularly hard" we just have
Lissa's question (as gist clause) "Did you find your favorite class
hard?", rather than "Did you find microeconomics hard?" -- in setting
up gist clauses for the main Lissa script, we have no way of knowing
in advance what the user's favorite class will be! It seems that we
need some way of allowing for parameters in Lissa outputs and in
gist clauses corresponding to those outputs, where these are filled
in as the dialog progresses.

Of course, there are other directions that are likely to become
important in the future, such as making inferences from user input
(via corresponding gist clauses, or ultimately wffs).