Massachusetts Institute of Technology
6.863J/9.611J, Natural Language Processing, Spring, 2003
Department of Electrical Engineering and Computer Science
Department of Brain and Cognitive Sciences

Laboratory 4: Semantic Interpretation with phrase structure grammars

Handed out: March 21, 2003 **Due: April 9, 2003**

1 Introduction: Goals of the Laboratory

As we've seen in lecture, broadly speaking there are two distinct current approaches to semantic interpretation, each with its own characteristics and (perhaps) applications.

Accordingly, In this laboratory you will design and write rules to extend a system that does semantic interpretation. The system works in conjunction with a context-free parser and the same kind of atomic nonterminal phrase structure grammar you wrote in Laboratory 3. When you have completed the assignment, you should understand the principles of compositionality in semantics and how that relates to syntactic analysis. And, you should know how to "put it all together": you will have extended an English interface to a simple database that will be able to either add facts to the database or retrieve answers from it:

```
athena>(top-level)
shall i clear the database? (y or n) y
sem-interpret>John saw Mary in the park
ok.

sem-interpret>where did john see mary
in the park.

sem-interpret>john gave fido to mary
ok.

sem-interpret>who gave john fido
i don't know

sem-interpret>who gave mary fido
john

sem-interpret>john saw fido
ok.

sem-interpret>who did john see
fido and mary
```

In general, this laboratory will involve a lot more reading and thinking than code writing. *Most* of the rules for the system have been already written; you will simply be adding some new ones to handle new sentence types, including some of those above (see section 3 below). For instance, the rules as given cannot handle *John gave Fido to Mary* or *Who gave Mary Fido*. Once you understand the basic idea behind the system, described in more detail in section 4, it is fairly straightforward to add new

capabilities; in fact, if you examine the existing system you will find that many of its capabilities have not been exploited in this laboratory. (See section 4 for more example traces.)

As usual, a completed laboratory will consist of a description and explanation of what you did to get your new rules to work, along with traces demonstrating that your new rules operate as advertised, and can reproduce the input-output described below. As usual, send a pointer to your web write-up.

In particular, this laboratory is designed to get you to understand the following four assumptions about the design of semantic interpreters, which have played a leading role in many contemporary approaches to sentence meaning (the method adopted here is one due essentially to R. Montague, but the same assumptions show up in many other systems):

1. The **rule-to-rule** principle of semantic interpretation: for each syntactic rule in the grammar, there is one rule of semantic interpretation. In the laboratory system this is implemented directly by pairing each context-free rule with a corresponding semantic rule (see section 4 below).

2. **Compositionality**: The meaning of a phrase is a function of the (meaning of) its parts, and nothing more. Again, this is implemented transparently by defining the "meaning" of, say, an S in the rule S→ NP VP as the result of the composition of the "meaning" of the VP and the "meaning" of the NP. In particular, we take the VP's meaning as supplying a procedure that is called with the "meaning" of the NP as its argument. (You might want to think about what sorts of sentences do *not* obey this principle, a question we raise again below.)

3. **Truth-conditional meaning**: the meaning of a sentence is equated with a set of **conditions** that make the sentence **true** or **false**. That is, it assumes that to know the meaning of a sentence is to know the set of conditions or the way the world would have to be in order for the sentence to be true—a relation between language and the world. Hence this approach adopts what is called a **correspondence theory** of truth. This is not a logically necessary component of a theory of meaning; there are alternatives such as the "mental picture" view or the "appropriate use" view of sentence meaning.

   In our laboratory's database world, the conditions for truth-correspondence are simple: the collection of "true facts" is represented by a set of (recursive) templates that stand for "event structures" of the form,

   ```
   (EVENT :condition1 val1 :condition2 val2 ... :conditionN valN)
   ```

   where an EVENT is just the main verb of a sentence and a constructed template is true iff it *matches* some template in this set (using a definition of *match* that is defined in the file `match-allowing-extras`). Note that we could also think of this as a thematic representation of the sentence.

4. **Model-theoretic semantics**: the correlation between language and the world can be explicated by building mathematical models of worlds using the machinery of set theory, and mapping linguistic expressions into them. Typically, lexical items are associated with set-theoretic objects. For example, we might associate the nouns of a language (*Mary*, *guy*) with the subsets of individuals in our model world that contains *Mary* or *guy*, etc. Syntactic rules for building phrases are associated with rules for combining set-theoretic objects. In our laboratory system, the rule-combining functions are built by constructing lisp procedures (in effect, the lambda-calculus).

It's easy to see why these assumptions have proved so popular with a database-oriented query language design: after all, we can view objects and relations in databases as just sets of objects. In this laboratory, it is interesting to see just how each of these assumptions gets cashed out.

2 Loading and running the system

First, attach 6.863 as usual, and then `cd` to the directory `semantics` under the course locker. Add Allegro common lisp via the following commands, and fire up Allegro common lisp via the command `mlisp` as shown. We have indicated how to use `mlisp` by running it under `xemacs` — this is much easier to use than just the command-line interface one would get with `mlisp` alone.

```
athena>add acl; mlisp-xemacs &
```

If one runs just the command `mlisp` alone, you will get back just the following immediately below. Otherwise, with the `mlisp-xemacs&` command, an emacs window should pop-up, that puts you in a Common Lisp listener environment, and then you should see the same sort of start up verbiage, but in this window. You proceed to load the semantic interpreter as shown. (We strongly recommend that you use xemacs in conjunction with Common Lisp.) If you've invoked `mlisp-xemacs` properly, In the xemacs buffer, you should see at the bottom the tag line `ACL Idle *common-lisp*` followed by `Inferior Common Lisp`, indicating that you are running Common Lisp from within xemacs.

```
*common-lisp*

=====================================================
Starting image ‘/mit/acl_v6.2/distrib/@sys/acl62/mlisp’
  with no arguments
  in directory ‘nil’
  on machine ‘localhost’.

International Allegro CL Enterprise Edition
6.2 [Solaris] (Jun 26, 2002 11:20)
Copyright (C) 1985-2002, Franz Inc., Berkeley, CA, USA.  All Rights Reserved.

This development copy of Allegro CL is licensed to:
   [TC3004] MIT Software Acquisition

;; Optimization settings: safety 1, space 1, speed 1, debug 2.
;; For a complete description of all compiler switches given the current
;; optimization settings evaluate (explain-compiler-settings).
;;---
;; Current reader case mode: :case-sensitive-lower
cl-user(1):
```

Now you can load the file `semantics.lisp`, which will set up the semantic interpreter. (This assumes you are already in the `/mit/6.863/semantics/` directory. If not, you have to give the full pathname.)

```
cl-user(1): (load "semantics")
Loading /afs/athena.mit.edu/course/6/6.863/semantics/semantics.lisp
;   Loading /mit/6.863/semantics/earley.lisp
;   Loading /mit/6.863/semantics/match-allowing-extras.lisp
;   Loading /mit/6.863/semantics/interpreter.lisp
;   Loading /mit/6.863/semantics/new-rules.lisp
;   Loading /mit/6.863/semantics/new-words.lisp
t
```

Now you can fire up the read-eval loop for semantic interpretation:

```
cl-user(2): (top-level)
shall i clear the database? (y or n) y
sem-interpret>john saw mary

syntax: (root
          (s (np (np-pro (name john)))
             (vp (v+args (v2+tns saw) (np (np-pro (name mary)))))))

semantics: (see :agent john :patient mary :tense past)

ok.

sem-interpret>did john see mary

syntax: (root
          (q-wh (aux%do%modal (aux%do+tns did))
           (np (np-pro (name john)))
           (vbar-tns (v2-tns see) (np (np-pro (name mary))))))

semantics: yes/no-question (see :agent john :patient mary :tense past)

yes.
sem-interpret>    [just a carriage return entered]

cl-user:(3):
```

To actually use the semantic interpreter, one can use either one of two modes: (1) enter a read-evaluate-print loop, via the procedure `(top-level)`; or (2) run through a batch list of test sentences via the procedure `test-semantics`.

If you use the first method, one can enter sentences one at a time, as shown below. Entering just a carriage return without any sentence, atlernatively, `quit`, will exit the read-eval-print loop and return you to common lisp's toplevel. Note that the first thing the system does is ask whether or not you want to clear the database of facts, stored in the global variable `*db*`; you enter `y` if you do, to start with a clean slate and `n` if not. If you don't clear the db, whatever values have been previously stored are carried over. Note also that no punctuation is used and capitalization is ignored.

To get rid of the `syntax:` and `semantics:` lines so that one just sees your input and the system's response, simply set the global variables `*show-syntax?*` and `*show-semantics?*` to `nil`:

```
cl-user(2): (setf *show-syntax?* nil *show-semantics?* nil)
NIL
cl-user(3): (top-level)
shall i clear the database? (y or n) y
sem-interpret>john saw mary

ok.
cl-user(4): (top-level)
Shall I clear the database? (y or n) n
sem-interpret>did john see mary

yes.
sem-interpret>
```

The second method of interaction requires that you bind a list of sentences to the global variable *semantics-tests*, and then call `test-semantics` (without any arguments). This leaves the fact-database untouched.

```
athena>(setf *semantics-tests* '("john saw mary" "did john see mary"))
("john saw mary" "did john see mary")

athena> (init-syntax)
athena> (test-semantics)
sentence: "john saw mary"

syntax:((root (s (np (np-pro (name john))) (vp (v+args (v2+tns saw)
                                  (np (np-pro (name mary))))))))
ok.
semantics: NIL

sentence: "did john see mary"
syntax: ((root (q-wh (aux%do%modal (aux%do+tns did))
                     (np (np-pro (name john)))
                     (vbar-tns (v2-tns see) (np (np-pro (name mary)))))))

yes.

athena>
```

Finally, there are three utility functions that will be especially valuable in doing the laboratory, but which are explained in section 4 below: `trace-semantics`, `untrace-semantics`, and `list-semantics`. These do what you think they do — turn on/off tracing of the semantic interpretation, and list the semantic interpretation rules themselves.

3 What you have to do for this lab

As it stands, the system uses the same style rules as in laboratory 3, coupled with the new semantic interpretation machinery. The interpreter first parses sentences using the parser as in Lab 3. Each syntactic and lexical rule is replaced with a *pair* of (syntactic, semantic) or (vocabulary, semantic) rules. The first element of each pair has the same format as in lab 3, while the second element is a lambda function, i.e., a procedure that will get invoked to determine the meaning associated with the lefthand side of the rule. Syntactic rules are added via the procedure `add-rule-sem`, one for each phrase structure rule; while vocabulary rules are added via the procedure `add-word-semantics`.

The resulting phrase structure(s), if any, are passed to the semantic interpreter. The interpreter uses the semantic interpretation principles 1–4 above, along with its semantic interpretation rules, to build a query/event **template**, which is used to either augment or retrieve information from a database of facts (stored in the global variable `*db*`).

The paired syntactic/semantic rules for phrases are in the file

`/mit/6.863/semantics/new-rules.lisp`

The paired syntax-semantics rules for words are in the file

`/mit/6.863/semantics/new-words.lisp`

(see section 4 for a description of their format). The existing system can process simple declarative sentences and simple yes-no questions.

Your job in this lab is to add new paired syntactic and semantic rules for phrases and new paired category and semantic vocabulary rules for words so that the system can correctly handle the following statements/queries.

You will only have to modify the `new-rules` *and* `new-words` *files in order to get this job done. Not many new rules and vocabulary words need to be added.* In particular, you should *not* have to modify any of the supporting code that carries out the database matches, spits out answers, and so forth (unless you want to do extra work, that is). Note that we *implicitly* provide the new *syntactic* rules you need by giving below the desired output parse trees for the new sentences to handle. (You can just read off the required syntactic rules from these.)

Your real work will be in writing/debugging the *semantic* rules for each syntactic rule so that the system gives back appropriate answers, as shown below (see section 4 for a description of what these rules look like). Full traces for these example sentences follow, showing the relevant parse tree and semantic query output (for additional explanation of the semantic query templates and other examples of the system in action see section 4 below). These sentences are also in the file `/mit/6.863/semantics/lab3-sentences.lisp`, already set up in such a way that they can be more easily run through the query interpreter. (We described above how to do this using `test-semantics`. The file has everything already set up to go.)

What you must do: Starting from the grammar supplied in the file `new-rules.lisp` here are the parse trees your final system should build and the resulting semantic interpretation query/fact templates and answers that a successful rule set should produce for these examples. You should follow the sentence order given below, so that you can be sure that the system gives the right answers to the questions that it's asked. Note that the parse trees output should immediately tell you what new phrase structure rules to add. (The challenge is adding the appropriate semantics.)

Hand in: Traces that match the syntactic and semantic interpretations shown here, along with the answers the system retrieves; your modified grammar rule and word rule files, along with brief comments about what you added and why.

IMPORTANT: Please read section 4, describing how the interpreter works, **before** attempting to add new rules.

In order to illustrate the steps you ought to go through in doing this, we show how to add the syntactic and semantic rules for dealing with the second example sentence in your assignment list below, *John gave Fido to Mary. to – prepositional phrase* type sentences, where the PP is an argument to a verb such as *give*.

We describe in turn:

- What syntactic rule to add
- What semantic rule(s) to add;
- What word rule(s) to add

You first should look at the syntactic parse tree and the desired semantic output:

```
sem-interpret>john gave fido to mary

syntax: (root
          (s (np (np-pro (name john)))
             (vp
              (v+args (v3+tns gave) (np (np-pro (name fido)))
               (pp+dat (p+to to) (np (np-pro (name mary))))))))

 semantics: (give :agent john :patient fido :beneficiary mary :tense past)

ok.
```

We are given a syntactic parse tree and the resulting output semantic "event template" with its values filled in. The event template is a (recursive) feature-value structure. What we don't show, and what you must be aware of, is the result of the function `process-sentence` on this meaning — it must either add some information to the database, retrieve it, etc. You should examine the code to see that this is essentially a case statement based on the type of sentence that is parsed, and we leave this to you. But onwards.

From the syntactic tree, it's straightforward to see that we must add the grammar rules to expand verbs like *give* into a verb of a certain class, namely, class 3, and two arguments: an NP and a PP of a special sort, PP+Dat. The "Dat" means "dative" verbs, like *give* that can appear in patterns like *gave Fido to John* or *gave John Fido*. Note that we need to distinguish the PP here with the attached tag "dat" so as ensure that the noun phrase part of this PP is taken as a real argument to the verb, rather than a PP such as "in the park" which is not an argument to a verb, as in "John saw Fido in the Park". One could also execute this via a feature-based grammar, but we don't do so here, and you will not need to in this laboratory either. In short, from the parse tree we can read off two new phrase structure rules that we must add:

```
v+args ==> v3+tns np pp+dat
pp+dat ==> p np
```

What about the corresponding semantic rules and words? We must pair a semantics rule for each of the two new syntactic rules, and also add new entries (perhaps) for word semantics. Our goal is to reach the semantic template displayed above, with the values filled in for agent, patient, and beneficiary. The agent is the "doer" of the sentence — the thing/person carrying out the action; the beneficiary is the recipient of the giving action; and the patient is the thing/item that is given. Before plunging ahead, you should note that this "meaning template" is already provided by the entry for *give* in the file `new-words.lisp`. Putting to one side the fluff that comes from the definitions in the file here that are associated with the use of a macro, this template for `v3+tns` is this:

```
'(lambda (agent beneficiary patient)
                `(,v-tns :agent ,agent :patient ,patient :beneficiary ,beneficiary
                                                         :tense past))
```

Note the use of the *backquote* symbol here before the second line. If you don't remember what backquote does, then recall that just like the quote symbok, this is used to *block* evaluation and simply pass along, everything inside the quoted expression, **but with important difference of allowing** the *selective*

evaluation of some items inside the quoted expression. (This is in contrast to the good-old-regular quote symbol which blocks evaluation of *everything* inside it). To evaluate any item inside a backquoted form, we put a comma before it. So you will note that we've put a quote before the verb `v-tns`, and the three arguments `agent`, `patient` and `beneficiary`. We have left the colon keywords `:agent`, `:patient`, `:beneficiary`, and `:tense` as is. Finally, we have supplied the actualy *value* to be associated with the tense keyword, namely, *past* — because for a verb `v3+tns` we know what the value should be. Thus, these colon keywords will wind up as part of the actual output template, as is, while everything else, aside from the value of the `:tense` keyword, will (eventually) have its value plugged in to give us the actual output semantic form:

```
(give :agent john :patient fido :beneficiary mary :tense past)
```

OK, with this goal in mind, let us consider the `pp+dat` semantics rule first. Note that what we want to do is have the noun phrase object of the Prepositional Phrase, e.g., "Mary" in "to Mary" passed on as the argument of the verb *give* that fills the position of the **beneficiary** (recipient) of give's action. We deduce this from looking at the semantic template. But this means we don't care about the preposition at all, and just want to "pass up" the semantic meaning (lambda form) associated with the NP. So our corresponding semantic rule should look like the following. It must be a lambda form of two arguments — the two nodes in the parse tree on the right-hand side of the context-free rule, p and np, and the semantic action should be just calling the function `p` on the argument (value of) `np`. That will ensure that, whatever the "value" of the noun phrase, it will get passed up to the `v+args` node. We call functions explicitly in lisp via the function `funcall` which takes the name of the function and its arguments.

```
'(lambda (p np)
          (funcall p np))
```

So, our complete new rule for `pp+dat`should look like this:

```
(add-rule-sem '(pp+dat ==> p np)
               '(lambda (p np)
                   (funcall p np)
```

This also tells us right away what the corresponding *semantics* should be for the corresponding word associated with the preposition *to*: since we just want to "pass up" the value of the noun phrase, the semantics of *to* can simply be the identity function. When the identity function is applied to the value of the noun phrase, it simply returns the value of the noun phrase as its result, just as desired:

```
(add-word-semantics 'to 'p 'identity)
```

And that is it for dealing with the dative prepositional phrase — just one semantic rule for the phrase, and one for the word entry for *to* (Yes: of course there is *another* semantic entry for *to* that corresponds to its meaning as a locational preposition — you might think about how to add that. But we don't do so in this lab.)

What about the semantic rule to pair with the expansion of `v3+args`? Again, to start out, just as with the `pp+dat` expansion, there are three phrases on the righthand side of the rule that expands `v3+args`, and so three arguments to the associated lambda form. So the associated semantic rule must begin:

```
'(lambda (v3+tns np pp+dat)
.......
```

But now we must do something a bit more subtle than just bluntly returning the straight-out value as before. What is the "value" (meaning) of a verb phrase? In the case of verb phrase expansions like this, recall that we want our function (our lambda form) to **return a function as its value** — that is, we want to *return* a lambda form. This is the "meaning" of the `v3+args` phrase, and in turn, the meaning of the VP, which will eventually be applied to the value of the Subject noun phrase.

In other words, what we want to *return* as the value of the lambda form for `v3+tns` is itself a lambda form, with a single argument, the subject of the sentence. So we should start out like this:

```
'(lambda (v3+tns np pp+dat)
                      `(lambda (subj).....
```

As before, here we need to use a backquote, because at this point in our assembly of the function calls to fill in the meaning template we don't want to apply the function now, but only when we have finally gotten a hold of the value (meaning) of the subject noun phrase, which we don't know at this point. On the other hand, we *do* know the meanings associated with the verb entry `v3+tns`, which has been provided to us already. So, we want the *value* of the semantic entry for `v3+tns`, which we can grab by putting a comma before its name —- this forces evaluation even behind the "shield" of a backquote, in contrast to a regular quotation symbol:

```
'(lambda (v3+tns np pp+dat)
                    `(lambda (subj)
                           (funcall ,v3+tns ....
```

Finally, the function associated with the verb entry for `v3+tns` must be called with its three arguments. Of these three arguments, the subject argument is not yet known at all, so we just leave it as a dummy variable at this point, `subj`. However, the two other arguments to the verb, the NP and the PP argument *are* known: they are the lambda forms passed up "from below". So, at first blush, one might think that the correct way to write the semantic rule for `v+args` would be as follows, with commas in front of the `pp+dat` and `np` arguments. This is almost correct, but not quite:

```
'(lambda (v3+tns np pp+dat)
                    `(lambda (subj)
                          (funcall ,v3+tns subj ,pp+dat  ,np))
```

What's wrong with this? Note we want to apply the function associated with `v3+tns` to the actual values of the arguments that follow. But if we put only commas in front of `pp+dat` and `np`, this will force evaluation of these forms as we desire, but it will go too far and leave the arguments as 'naked' values that the Lisp interpreter will (incorrectly) try to further evaluate as functions. To see this, consider, for example, the NP *Mary* in "give Fido to Mary." Our previous rule for `pp+dat` will give the semantics for the NP object as simply the constant `Mary`. So, if we retrieve this value via the lisp code `,pp+dat`, this will first return the form `Mary` **but then incorrectly** try to evaluate the constant `Mary` as if it were a variable with a value. Since `Mary` does not have any value besides itself, this would cause an error. Clearly, what we want to do is to put a quote in front of `Mary` so that its literal value is what is returned. Similarly for the other argument, `Fido`. Thus, what we really want is this, where we have put quote marks in front of the two arguments corresponding to the dative NP object and the second NP object

```
'(lambda (v3+tns np pp+dat)
                  `(lambda (subj)
                        (funcall ,v3+tns subj ',pp+dat  ',np))
```

This is the correct form. So, in all, we had to add two new paired syntax/semantics rules, one for the expansion of a dative `pp`, and one for the expansion of a dative verb. We had to add just one new lexical entry, for *to*. And now we're done. Note that we don't have to add anything at all to handle new words for NPs, since this is already taken care of by the semantics we need for NPs anyway. That is the beauty of decomposition. In general, you will find that this is typical: you should not have to add very many new rules or word entries to handle each new sort of sentence.

One more word of advice before moving on to the example sentences themselves. Note that the pattern-matcher used to match and retrieve facts from the database is very very over-simplified. Consider an event structure for, e.g., *John saw Mary in the park*, where we have added a sub-part `:location` to the event structure to correspond to the location *in the park*:

```
(see :agent john :patient mary :tense past :location (park :definite? t
                                                          :number   singular))
```

In order that the system can still get a partial match for a question like, *Did John see Mary?*, which would have only the more general event structure,

```
(see :agent john :patient mary :tense past)
```

the procedure `match-allowing-extras` allows the previous event structure to match the one directly above — as its name implies. However, this function is very very stupid, and requires that the elements that do match be present in the SAME order in both event structures. (Obviously this can be repaired — a topic for final projects: one should sort the keywords...). So, you must just take care here, otherwise, the matching/retrieval won't work properly.

The example sentences you must handle in this laboratory

Please note the use of lower-case for word input.

1. john gave mary fido (this is already handled!)
2. john gave fido to mary
3. did john give fido to mary
4. did john give mary fido
5. mary kissed john
6. fido was kissed
7. john was kissed by mary
8. john saw fido (this is already handled!)
9. john saw mary (as is this!)

10. who saw fido
11. who did john see
12. who gave mary fido
13. who gave john fido
14. who gave fido to mary
15. who did john give fido to
16. who did john give to mary
17. who kissed fido
18. who did mary kiss
19. who was kissed by mary
20. did somebody kiss fido

There are *three* basic classes of sentences to be added for semantic interpretation: (1) rules to handle to- PPs after verbs like *give* — this we've already shown you how to do; (2) passive sentences; and (3) wh-questions beginning with *who*. In order to actually add in new rules, you will have to modify two global variables:

`*word-semantics-file*`

and

`*rule-semantics-file*`

so their values match the names of your own files for word and rule definitions, e.g.,

```
(setf *rule-semantics-file* "/mit/mydir/my-rule-file")
(setf *word-semantics-files* "/mit/mydir/my-word-vocabulary-file")
```

These variables are used by the routine `init-syntax` which does the actual work of reading in the grammar rules, building the Earley-style table and the associated semantic interpretation rules, and then running the syntax parser on a simple test sentence. If you change the values of `*rule-semantics-file*` and `*word-semantics-file` then `init-syntax` will read in your new definitions. The way the system is set up these will have to be copies of whatever is in `new-rules.lisp` and `new-words.lisp` *plus* any new rules that you have added, residing in your own directory as shown above. The format for new rules is described below in section 4.

Traces of the lab example sentences

```
;;; Examples for lab
sem-interpret>>john gave mary fido
syntax: (root
          (s (np (np-pro (name john)))
             (vp (v+args (v3+tns gave) (np (np-pro (name mary)))
             (np (np-pro (name fido)))))))
semantics: (give :agent john :patient fido :beneficiary mary :tense  past)
ok.
```

```
sem-interpret>john gave fido to mary

syntax: (root
          (s (np (np-pro (name john)))
             (vp
              (v+args (v3+tns gave) (np (np-pro (name fido)))
               (pp+dat (p+to to) (np (np-pro (name mary))))))))

 semantics: (give :agent john :patient fido :beneficiary mary :tense past)

ok.

sem-interpret> did john give fido to mary

syntax: (root
           (q-wh (aux%do%modal (aux%do+tns did))
                 (np (np-pro (name john)))
                 (vbar-tns (v3-tns give) (np (np-pro (name fido)))
                    (pp+dat (p+to to) (np (np-pro (name mary)))))))

semantics: yes/no-question (give :agent john :patient fido :beneficiary mary :tense past)
yes.

sem-interpret> did john give mary fido

syntax: (root
           (q-wh (aux%do%modal (aux%do+tns did))
                 (np (np-pro (name john))) (vbar-tns (v3-tns give)
                                                     (np (np-pro (name mary)))
                                                     (np (np-pro (name
  fido))))))

semantics: yes/no-question (give :agent john :patient fido  :beneficiary mary :tense past)
yes.


sem-interpret> mary kissed john

syntax: (root
           (s (np (np-pro (name mary)))
              (vp (v+args (v2+tns kiss) (np (np-pro (name john)))))))

semantics: (kiss :agent mary :patient john :tense past)

ok.
```

```
sem-interpret>fido was kissed

syntax: (root
           (s (np (np-pro (name fido)))
              (vp (aux%be+tns was) (v+passp (v2+passp kissed)))))

semantics: (kiss :agent somebody :patient fido :tense past :type passive)
ok.

sem-interpret> john was kissed by mary

syntax: (root
           (s (np (np-pro (name john)))
              (vp (aux%be+tns was) (v+passp (v2+passp kissed)
              (pp+agby (p by) (np (np-pro (name mary))))))))

semantics: (kiss :agent mary :patient john :tense past :type passive)

ok.

sem-interpret> john saw fido

syntax: (root
           (s (np (np-pro (name john)))
              (vp (v+args (v2+tns saw) (np (np-pro (name fido)))))))

semantics: (see :agent john :patient fido :tense past)

ok.

sem-interpret>john saw mary

syntax: (root
           (s (np (np-pro (name john)))
              (vp (v+args (v2+tns saw) (np (np-pro (name mary)))))))

semantics: (see :agent john :patient mary :tense past)

ok.

sem-interpret> who saw fido

syntax: (root
           (q+wh (np+wh (pronp+wh who))
                 (vp (v+args (v2+tns saw) (np (np-pro (name fido)))))))

semantics: wh-question ?who (see :agent ?who :patient fido :tense past)

john
```

```
sem-interpret> who did john see

syntax: (root
           (q+wh (np+wh (pronp+wh who))
                 (q/np (aux%do%modal (aux%do+tns did))
                 (np (np-pro (name john)))
                 (vbar-tns/np (v2-tns see) (np/np)))))

semantics: wh-question ?who (see :agent john :patient ?who :tense past)

fido and mary

sem-interpret>who gave mary fido

syntax: (root
           (q+wh (np+wh (pronp+wh who))
                 (vp (v+args (v3+tns gave) (np (np-pro (name mary)))
                                           (np (np-pro (name fido)))))))

semantics: wh-question ?who (give :agent ?who :patient fido :beneficiary mary :tense past)

john

sem-interpret>who gave john fido

syntax: (root
           (q+wh (np+wh (pronp+wh who))
                 (vp (v+args (v3+tns gave) (np (np-pro (name john)))
                                           (np (np-pro (name  fido)))))))

semantics: wh-question ?who (give :agent ?who :patient fido  :beneficiary john :tense past)

i don't know.

sem-interpret>who gave fido to mary

syntax: (root
           (q+wh (np+wh (pronp+wh who))
                 (vp (v+args (v3+tns gave) (np (np-pro (name fido)))
                                           (pp+dat (p+to to) (np (np-pro (name mary))))))))

semantics: wh-question ?who (give :agent ?who :patient fido  :beneficiary mary :tense past)

john
```

```
sem-interpret>who did john give fido to

syntax: (root
           (q+wh (np+wh (pronp+wh who))
                 (q/np (aux%do%modal (aux%do+tns did))
                 (np (np-pro (name john)))
                 (vbar-tns/np (v3-tns give) (np (np-pro (name fido)))
                                            (pp+dat/np (p+to to) (np/np))))))

semantics: wh-question ?who (give :agent john :patient fido :beneficiary ?who :tense past)

mary

sem-interpret>>who did john give to mary

syntax: (root
           (q+wh (np+wh (pronp+wh who))
                 (q/np (aux%do%modal (aux%do+tns did))
                 (np (np-pro (name john)))
                 (vbar-tns/np (v3-tns give) (np/np) (pp+dat (p+to to)
           (np (np-pro (name mary)))))))

semantics: wh-question ?who (give :agent john :patient ?who  :beneficiary mary :tense past)

fido

sem-interpret>who kissed fido

 syntax: (root
          (q+wh (np+wh (pronp+wh who))
           (vp (v+args (v2+tns kissed) (np (np-pro (name fido)))))))

semantics: wh-question ?who (kiss :agent ?who :patient fido :tense past)

somebody

sem-interpret>who did mary kiss

syntax: (root
           (q+wh (np+wh (pronp+wh who))
                 (q/np (aux%do%modal (aux%do+tns did))
                 (np (np-pro (name mary))) (vbar-tns/np (v2-tns kiss) (np/np)))))

semantics: wh-question ?who (kiss :agent mary :patient ?who :tense past)

john
```

```
sem-interpret>who was kissed by mary

syntax: (root
           (q+wh (np+wh (pronp+wh who))
                 (vp (aux%be+tns was) (v+passp (v2+passp kissed)
                                               (pp+agby (p by) (np (np-pro (name mary))))))))

semantics: wh-question ?who (kiss :agent mary :patient ?who :tense past :type passive)

john

sem-interpret>did somebody kiss fido

syntax: (root
           (q-wh (aux%do%modal (aux%do+tns did))
                 (np (np-pro (name somebody)))
                 (vbar-tns (v2-tns kiss) (np (np-pro (name fido))))))

semantics: yes/no-question (kiss :agent somebody :patient john :tense past)

yes.
```

4 The semantic interpreter

In this section we describe the format for semantic interpretation rules and how the interpreter actually does its job. It is important to understand this material in order to add new rules.

Recall first that the job of semantic interpretation is to build a thematic or event representation, a *frame*, of the sentence. The frame can be used to access a database, either in the form of a *fact* or a *query*. A *fact* will look something like this, where the first item in the list corresponds to the main predicate of the sentence, and the remaining items are pairs of thematic roles/attributes paired with values; the structures here may be recursive. The thematic representation is primitive, but expandable; it includes the usual notions of AGENT, PATIENT (affected object), BENEFICIARY (recipient of object), LOCATIVE, MOOD, TYPE, etc.; see the `new-words.lisp` file for more details.

A fact is simply stored in the database, and this is the result of semantic interpretation in this case. For instance, from *John saw Mary* we would get the following interpretation or fact to be added to the database:

```
(see :agent john :patient mary :tense past)
```

A query is either a Yes/No question or a WH-question. If it's a Yes/No question, we simply wrap a procedure called `y/n-question` around the fact returned by the remainder of the sentence; this procedure simply queries the database about the fact asserted in the rest of the sentence. For example, *Did John see Mary* would simply query the database about the same fact as asserted above (adding the `:tense` marker via the word *did*).

For a wh-question, we wrap a procedure called `wh-question` around the the pieces of the resulting frame. This procedure takes two arguments: a wh-variable, and a frame with that wh-variable filling the place of the wh-element that is questioned. Here's an example from the test sentences, where *who* is the questioned item:

```
sem-interpret>who kissed fido

syntax: (root
          (q+wh (np+wh (pronp+wh who))
           (vp (v+args (v2+tns kissed) (np (np-pro (name fido)))))))

semantics: wh-question ?who (kiss :agent ?who :patient fido :tense past)
```

The matcher is smart enough not to care about extraneous details in answering queries. For example, if given the query,

```
(chase :agent mary :patient ?who :tense past)
```

then this will match against a previously stored event that includes a `:location` value. You shouldn't have to alter any of this machinery.

We shall see how this is all put together shortly. First however we describe the format for grammar rule and dictionary entries.

Rule formats. This lab replaces each syntactic and vocabulary rule with a *pair* of (syntactic, semantic) or (vocabulary, semantic) rules. The first element of each pair has the same format as in lab 3, while the second element is a lambda function, i.e., a procedure that will get invoked to determine the meaning associated with the lefthand side of the rule. Syntactic rules are added via the procedure `add-rule-sem`, one for each phrase structure rule; while vocabulary rules are added via the procedure `add-word-semantics`.

Here are some examples. Let us look at a rule to process a simple phrase structure rule, S$rightarrow$NP VP. The format is simple. We first include the phrase structure expansion just as in lab 3, and follow it with a lambda procedure whose arguments include just the two phrase structure nodes mentioned on the righthand side of the rule. The system automatically uses the single procedure `add-rule-sem` to add both a syntactic context-free rule and an associated semantic interpretation rule to the system, one at a time. (Thus one does not add in a whole list of syntactic rules at a time, as in lab 3.)

Note that this format automatically follows the first two principles mentioned earlier: the rule-to-rule hypothesis (for each syntactic rule there is a corresponding rule of semantic interpretation) and compositionality (the meaning of a phrase like S is purely a function of the meaning of its parts NP and VP).

People will note that this lambda procedure simply says to call the procedure named `vp` (whose value or "meaning" will be determined "from below", that is, from the semantic Interpretation of the VP node) using the argument `np` (whose meaning will again be determined from semantic interpretation further down the tree subsuming NP). In other words, to find the interpretation of the S, we call the procedure VP using as an argument the subject NP. These two values will be supplied by the (recursive) semantic interpretation of the NP and VP nodes.

```
(add-rule-sem '(s ==> np vp)
                    '(lambda (np vp)
                        (funcall vp np)))
```

To finish off the description of rule formats, at the very bottom individual words must also contain some paired semantic "value" (in fact, words with multiple meanings could have more than one semantic value). In general words create thematic frames or add entries to an existing frame. For individual names, this is easy: the proper set to return is just the individual label itself (thus fixing a pun between the name of the token *John* and its semantic value in the database). Thus the first two components of `add-word-semantics` looks just as in lab 3, listing the word and its syntactic category. The third

component is its paired semantic value. Words that are to be ignored in the semantics (but play some syntactic role) can just be given the identity function as their semantic value:

```
(add-word-semantics 'John 'name 'John)

(add-word-semantics 'that '*that '(lambda ()))
```

Multiple entries are permitted, as before, as well as multiple "meanings."

Verbs have more complicated semantic entries. For instance, *did* requires an associated procedure that will *add* to the frame already constructed; we saw that this was necessary for a yes-no question, for example. Here is the relevant lexical entry. It has an associated lambda procedure that takes as its sole argument the frame constructed to the right of *did* (associated with either an S or a VP node) and adds a tense marker to that frame:

```
(add-word-semantics 'did 'aux%do+tns
                    '(lambda (verb-frame)
                       (append verb-frame '(:tense past))))
```

You will have to add new entries of both kinds, with both syntax to both the words and rules files. A few macros have been supplied to make adding new verb and noun entries easier (`add-v1, add-v2, add-noun`).

How it all fits together

The overall design of the interpreter is to walk recursively down a completed phrase structure tree via the procedure `phrase-semantics`, pasting together the meanings of a whole phrase out of the meanings of its parts. At each step, we call either `rule-semantics` (if the node is a phrase) or `word-semantics` (if the item is a leaf node). Of course, we must actually execute the procedures attached to each node; this is done by the procedure `rule-apply`. Thus the entire system is analogous to Lisp's `eval` and `apply`; in general, `rule-apply` must be called once for each branching rule in the phrase structure tree.

Let's see how this all fits together in a simple example. In general, such a system will dive all the way down to the leaves of a tree and then glue the required meanings together via composed lambda procedures on the way back out again. (A full trace of this example is given immediately below; here we'll go through things step by step.) In an actual declarative sentence example, say, *John saw Mary*, the parser first produces the phrase structure,

```
(ROOT
  (S (NP (NP-PRO (NAME JOHN)))
     (VP (V+ARGS (V2+TNS SAW)
         (NP (NP-PRO (NAME MARY)))))))
```

The semantic interpreter now calls its main routine, `phrase-semantics`, on this tree, recursively traversing it depth-first.

The topmost ROOT node will have an associated lambda procedure that simply calls the function `process-sentence`:

```
(add-rule-sem '(root ==> s)
                    '(lambda (s)
                        (process-sentence s)))
```

Thus, in the end this rule will take this lambda form, plus whatever is returned by the semantic interpretation of S, namely a frame, as an argument. Using `rule-apply`, the system will put in the semantic interpretation of S as the value of `s`—the argument to `process-sentence`. This will simply

add the frame as a fact to the database. (All `process-sentence` does is call a procedure `db-add` with the frame as its argument, which in turn just glues the fact on the front of the database.)

```
(lambda (s) (process-sentence s)
       '(see :agent john :patient mary :tense past))
```

So how does S get its interpretation (a frame)? We saw above that the Interpretation of S is just the result of calling VP with an argument that's the subject NP:

```
(lambda (np vp)
    (funcall vp np))
```

This procedure needs two values: one for the VP and one for the NP. These two values are filled in from below, and substituted via the evaluation done by `rule-apply`. Let's see where the values come from.

The Subject NP is built syntactically as,

```
(np (np-pro (name john)))
```

If we look at the three rules for these items, NP, NP-PRO, and NAME, we find:

```
(add-rule-sem '(np ==> np-pro) #'identity)

(add-rule-sem '(np-pro ==> name) #'identity)

(add-word-semantics 'John 'name 'John)
```

So the calls to `phrase-semantics` here just lead to the composition of two identity functions followed by the constant `John`, i.e., just the constant `John`. This is confirmed by the trace:

```
|   3 Enter PHRASE-SEMANTICS (NP (NP-PRO (NAME JOHN)))
|   | 4 Enter RULE-SEMANTICS NP (NP-PRO)
|   | 4 Exit RULE-SEMANTICS #<Compiled-Function IDENTITY 17EBFF>
|   | 4 Enter PHRASE-SEMANTICS (NP-PRO (NAME JOHN))
|   |   5 Enter RULE-SEMANTICS NP-PRO (NAME)
|   |   5 Exit RULE-SEMANTICS #<Compiled-Function IDENTITY 17EBFF>
|   |   5 Enter PHRASE-SEMANTICS (NAME JOHN)
|   |   | 6 Enter WORD-SEMANTICS JOHN NAME
|   |   | 6 Exit WORD-SEMANTICS JOHN
|   |   5 Exit PHRASE-SEMANTICS JOHN
|   | 4 Exit PHRASE-SEMANTICS JOHN
|   3 Exit PHRASE-SEMANTICS JOHN
```

The more interesting part is the semantic interpretation of the VP node. It is the key to understanding the Montagovian style of doing semantic interpretation.

Here is the syntactic rule associated with the VP expansion:

```
(vp ==>  v+args)
```

Here is the semantic lambda procedure associated with the VP rule:

```
(lambda (v+args)
    `(lambda (subj)
        (funcall ,v+args subj)))
```

Note that this is a procedure that itself returns a procedure—namely, that procedure object which takes one argument, the (semantic value of) the Subject, and calls the function produced by `V+args` on it. (Aside for novices: the comma `,` before `v+args` ensures that it is the value of `v+args` that is used.) Note that the actual semantic value for subject will be duly supplied higher up when we carry out the `funcall` of `vp` on `np`. This is typical of the procedures associated with VPs, PPs, AUXiliary nodes, and so forth in Montagovian semantics. For example, the semantics of the preposition *in* is a function tha takes as input a PP-complement (usually an NP) and returns a function that takes as input a thematic/event frame and adds the appropriate feature to that frame. (It's all part of the Montagovian cuisine: curried lambdas, or should we say, digested lambda's?)

Pasting together what we have so far then, since the semantic value of `VP` was `funcall`ed with the semantic value of `NP`, we have this:

```
(lambda (s) (process-sentence s)
   (lambda (subj) (funcall <lambda procedure produced by v+args> subj)
       'John))
```

Of course, this naturally leads to the question of what the procedure `v+args` should look like. You should be able to guess: for each verb subcategory, it constructs a lambda procedure which is a function of the verb and its arguments (excluding the subject); this lambda procedure in turn constructs the basic thematic frame whose values are filled in by the subject and the arguments to the verb. So for example, for *see* we need this syntactic rule:

```
(v+args ==> v2+tns np)
```

along with this semantic rule:

```
(lambda (v2+tns np)
       `(lambda (subj)
          (funcall ,v2+tns subj ',np)))
```

Note what this lambda form says: it expects two arguments to follow, (1) the semantic value of `v2+tns` and (2) the semantic value of an np (the object np). It *returns* a procedure with one argument as its value, and the values of `v2+tns` and `np` plugged in.

Remember that the value of `subj` will be filled in by lambda-substitution; however, the value of the direct object argument to the verb must be "passed up" as the value associated with the object NP node (that's why we've written the inner lambda using the `',` notation, which will insert the semantic value of the object NP into the `funcall`—it ought to be *Mary* in our example). What about the value of `v2+tns`? That is the verb subcategory corresponding to `see`, and so, at bottom, the meaning of this single word is just this, where the two arguments of `agent` and `patient` are filled in when the `funcall` is done by the `subj` and semantic value of the object NP. But this value will just be the constant *Mary* (analogously to how *John* was computed); we omit these steps.

```
(lambda (agent patient)
 `(see :agent ,agent :patient ,patient :tense past))
```

Note how this meshes with the `v+args` semantic rule, which said that the first argument to `v2+tns` should fill the agent role and is the subject, while the second argument is the np direct object and should come immediately following `v2+tns` (as it does, since the NP follows). The subject fills the `:agent` thematic role and the direct object NP fills the `:patient` role.

Taking stock of where we are so far then, when we substitute for the `v2+tns` and `np` components we get the semantic value produced by `v+args`, or effectively, the entire S form (omitting some details):

```
(lambda (subj)
  (funcall
      (lambda (subj)         ; this is the value of VP
              (funcall (lambda (agent patient)
                 `(see :agent ,agent :patient ,patient :tense past))
                  subj       ; agent argument supplied by lambda
  application
                  'Mary))    ; patient argument (semantic value of
  object np)
       subj))
```

applied to the subject argument `'John`. This gives us the desired frame to be used as an argument to `process-sentence`.

For the curious, the entire trace using the supplied procedure `trace-semantics` looks like this:

```
> (trace-semantics)
(WORD-SEMANTICS)
> (top-level)
Shall I clear the database? (Y or N) y
>john saw mary
1 Enter PHRASE-SEMANTICS (ROOT (S (NP #) (VP #)))
| 2 Enter RULE-SEMANTICS ROOT (S)
| 2 Exit RULE-SEMANTICS #<Interpreted-Function
                           (LAMBDA (S) (PROCESS-SENTENCE S)) 6C7DBF>
| 2 Enter PHRASE-SEMANTICS (S (NP (NP-PRO #)) (VP (V+ARGS # #)))
|   3 Enter RULE-SEMANTICS S (NP VP)
|   3 Exit RULE-SEMANTICS #<Interpreted-Function
                         (LAMBDA (NP VP) (FUNCALL VP NP)) 6C802F>
|   3 Enter PHRASE-SEMANTICS (NP (NP-PRO (NAME JOHN)))
|   | 4 Enter RULE-SEMANTICS NP (NP-PRO)
|   | 4 Exit RULE-SEMANTICS #<Compiled-Function IDENTITY 17EBFF>
|   | 4 Enter PHRASE-SEMANTICS (NP-PRO (NAME JOHN))
|   |   5 Enter RULE-SEMANTICS NP-PRO (NAME)
|   |   5 Exit RULE-SEMANTICS #<Compiled-Function IDENTITY 17EBFF>
|   |   5 Enter PHRASE-SEMANTICS (NAME JOHN)
|   |   | 6 Enter WORD-SEMANTICS JOHN NAME
|   |   | 6 Exit WORD-SEMANTICS JOHN
|   |   5 Exit PHRASE-SEMANTICS JOHN
|   | 4 Exit PHRASE-SEMANTICS JOHN
|   3 Exit PHRASE-SEMANTICS JOHN
|   3 Enter PHRASE-SEMANTICS (VP (V+ARGS (V2+TNS SAW) (NP #)))
|   | 4 Enter RULE-SEMANTICS VP (V+ARGS)
|   | 4 Exit RULE-SEMANTICS #<Interpreted-Function
            (LAMBDA (V+ARGS) (SYSTEM:BQ-LIST (QUOTE LAMBDA)
                             (QUOTE #) (SYSTEM:BQ-LIST* # V+ARGS #)))
                             6C8B17>
|   | 4 Enter PHRASE-SEMANTICS (V+ARGS (V2+TNS SAW) (NP (NP-PRO #)))
|   |   5 Enter RULE-SEMANTICS V+ARGS (V2+TNS NP)
|   |   5 Exit RULE-SEMANTICS #<Interpreted-Function
              (LAMBDA (V2+TNS NP) (SYSTEM:BQ-LIST (QUOTE LAMBDA)
                                  (QUOTE #) (SYSTEM:BQ-LIST # V2+TNS
                                  # #))) 6C8DE7>
|   |   5 Enter PHRASE-SEMANTICS (V2+TNS SAW)
|   |   | 6 Enter WORD-SEMANTICS SAW V2+TNS
|   |   | 6 Exit WORD-SEMANTICS #<Interpreted-Function
                 (LAMBDA (AGENT PATIENT) (SYSTEM:BQ-LIST*
                         (QUOTE SEE) :AGENT AGENT :PATIENT PATIENT
                         (QUOTE #))) 6C90D7>
|   |   5 Exit PHRASE-SEMANTICS #<Interpreted-Function
                       (LAMBDA (AGENT PATIENT) (SYSTEM:BQ-LIST*
                          (QUOTE SEE) :AGENT AGENT :PATIENT PATIENT
                          (QUOTE #))) 6C90D7>
|   |   5 Enter PHRASE-SEMANTICS (NP (NP-PRO (NAME MARY)))
|   |   | 6 Enter RULE-SEMANTICS NP (NP-PRO)
|   |   | 6 Exit RULE-SEMANTICS #<Compiled-Function IDENTITY 17EBFF>
|   |   | 6 Enter PHRASE-SEMANTICS (NP-PRO (NAME MARY))
|   |   |   7 Enter RULE-SEMANTICS NP-PRO (NAME)
|   |   |   7 Exit RULE-SEMANTICS #<Compiled-Function IDENTITY 17EBFF>
|   |   |   7 Enter PHRASE-SEMANTICS (NAME MARY)
```

```
|   |   |   | 8 Enter WORD-SEMANTICS MARY NAME
|   |   |   | 8 Exit WORD-SEMANTICS MARY
|   |   |   7 Exit PHRASE-SEMANTICS MARY
|   |   | 6 Exit PHRASE-SEMANTICS MARY
|   |   5 Exit PHRASE-SEMANTICS MARY
|   | 4 Exit PHRASE-SEMANTICS #<Interpreted-Function (LAMBDA (SUBJ)
            (FUNCALL #<Interpreted-Function (LAMBDA (AGENT PATIENT)
                  (SYSTEM:BQ-LIST*
                      (QUOTE SEE) :AGENT AGENT :PATIENT PATIENT (QUOTE
                      #)))
                               6C90D7> SUBJ (QUOTE MARY))) 6C9F67>
|   3 Exit PHRASE-SEMANTICS #<Interpreted-Function
                   (LAMBDA (SUBJ) (FUNCALL #<Interpreted-Function
                          (LAMBDA (SUBJ) (FUNCALL
                          #<Interpreted-Function
                              (LAMBDA (AGENT PATIENT)
                                 (SYSTEM:BQ-LIST*
                                    (QUOTE SEE) :AGENT AGENT :PATIENT
                                    PATIENT
                                       (QUOTE #))) 6C90D7> SUBJ
                                          (QUOTE MARY))) 6C9F67>
                                          SUBJ)) 6CA2AF>
| 2 Exit PHRASE-SEMANTICS (SEE :AGENT JOHN :PATIENT MARY :TENSE PAST)
OK.
1 Exit PHRASE-SEMANTICS NIL
```

Some additional helpful tips

Here are some other points to keep in mind when using the system.

- The procedure `list-semantics` can be used to print out in alphabetized, readable form the paired syntactic/semantic rules and the dictionary categories. An excerpt:

```
athena> (list-semantics)

(NP ==> NP-PRO) : IDENTITY

(S ==> NP VP) : (LAMBDA (S) (FUNCALL VP NP))

   etc.

(A . DET)
(AN . DET)

   etc.
(YELLOW . A)
```

- The tracing procedure `trace-semantics` just traces calls to `phrase-semantics`, `rule-semantics`, and `word-semantics`.
- The system can handle modals, adjectives, embedded questions, and other syntactic constructs not otherwise described here—take a look and find out for yourself!
- In order to put new syntactic rules into the system one has to call `init-syntax` all over again, because the entire Earley parsing table must be rebuilt. *However*, if one is changing only the semantic part of a rule, then one need only re-evaluate that single rule, because that lambda form will be placed into a lookup table immediately replacing the old version. Thus the semantics can be debugged incrementally if the syntax is unchanged.
- The routines for generating output from the query/facts are pretty simple-minded.