HP-NL Users Guide

Natural Language Project
Hewlett-Packard Laboratories

Documentation version 3.0
January 29, 1988

Chapter 13

Semantics Processor

13.1 Overview

The HP-NL semantics modules convert the parse trees produced by the parser into application language queries. The conversion is effected in four steps: *Initial Semantics*, the derivation of a context- and domain-independent logical representation; *Pragmatics Processor*, which anchors context parameters; *Disambiguation*, the extraction of a domain-specific representation; and *Transduction*, a transformation into a database query. This chapter describes the first of these steps.

The division of labor reflects varying degrees of (database-) independence of the three modules. The initial semantics is formulated in the logical language *NFLT*, described in Chapter 9. This representation is general and doesn't vary from one application to the next. A new application may emphasize a novel facet of this representation, or even call for innovations or refinements, but there should be no need to tailor the representation to accommodate different applications. The conversion from parse trees to NFLT is effected in the *Semantics Processor* and is described below; like NFLT itself, this module is constant from application to application.

The other parts of semantics processing cannot be completely application-independent. The application language may vary; we currently support HP-RL a frame representational language, and SQL, a standard query language for relational databases. (There is also a

more experimental interface to Prolog.) The intention of the division of labor here is that a single transducer should suffice for each application language, i.e. that further HP-NL databases should employ the same transducer. Finally, the intermediate module, *Disambiguation*, has to deal with differences that may arise even in applications using the same application language. Its task may best be clarified by an illustration. We may use the single English verb, *work*, in several different ways:

Is Jones's system working?
Who works for Jones?
Who does Jones work for?

We may not assume that all the information concerning *work* is stored in the same way. Whether or not a system is working is probably stored as a yes/no or boolean, while the information about who Jones works for has to be in the form of a personal ID. This is probably obvious. The second and third examples above likewise ask for information that is stored in different ways in the personnel database currently connected to the natural language system. In order to conserve space, the database records a *supervisor* under each employee, but does not record *subordinates* even if the employee is a supervisor. To find out who Jones works for, it is enough to look up *supervisor* under *Jones*, but to find out who works for Jones, a search is required to find out where *Jones* is listed as *supervisor*. In applications where speed was essential, another decision might have been made. It isn't the task of the natural language system to make or even anticipate these decisions, but they have to be dealt with, and this is the task of the disambiguator. For each application, the disambiguator must be provided with information about which database relations may be denoted by which application-independent NFLT relations.

Summing up, the natural language system completely isolates the *semantics processor* and *pragmatics processor* from the form in which information is stored, and furthermore isolates the module transducing from application-specific logical representation to database language. Linking the pragmatics processor and the transduction mondule, and accommodating specific decisions on how data is organized in the database, is the disambiguation module. The table below sketches these relationships and indicates the location of the

code performing the tasks.

Module	Input	Output	Code
NFLT Definitions	***	***	$nlsem/nflt.l
Initial Semantics	parse trees	NFLT formulæ	$nlsem
Disambiguation	NFLT formulæ	Mixed DB Lg./NFLT	$nldis/
Transducer	Mixed DB Lg./NFLT	DB Lg. (HP-RL)	$orgdb/db-htran.l/

13.2 Semantics Data Structures

13.2.1 Task of Semantics Processor

The semantics module takes the parse trees of natural language sentences as its input, transduces these to trees of semantics nodes and produces as output formulas in a logical language, NFLT, which represent the meanings of the sentences. The module is organized so that semantics may be computed for any parse tree, including trees of subsentential constituents. This is intended to allow semantics computation to follow after all parsing or to proceed in parallel with it. The current default for the system causes semantics to be computed following completion of all parsing.

13.2.2 Justification for Special Structures

In the present configuration, the semantics module transduces a tree of semantics data structures from the parse chart. Since all the information required for semantic processing is present in the parse chart, the question arises as to why we bother to create special data structures for semantics processing. The transduction is performed for two reasons: (1) it insulates the semantics and parsing components from each other, contributing to the maintainability and modifiability of the separate components; and (2) the transduction routine defines the interaction of syntactic and semantic information, contributing to transparency.

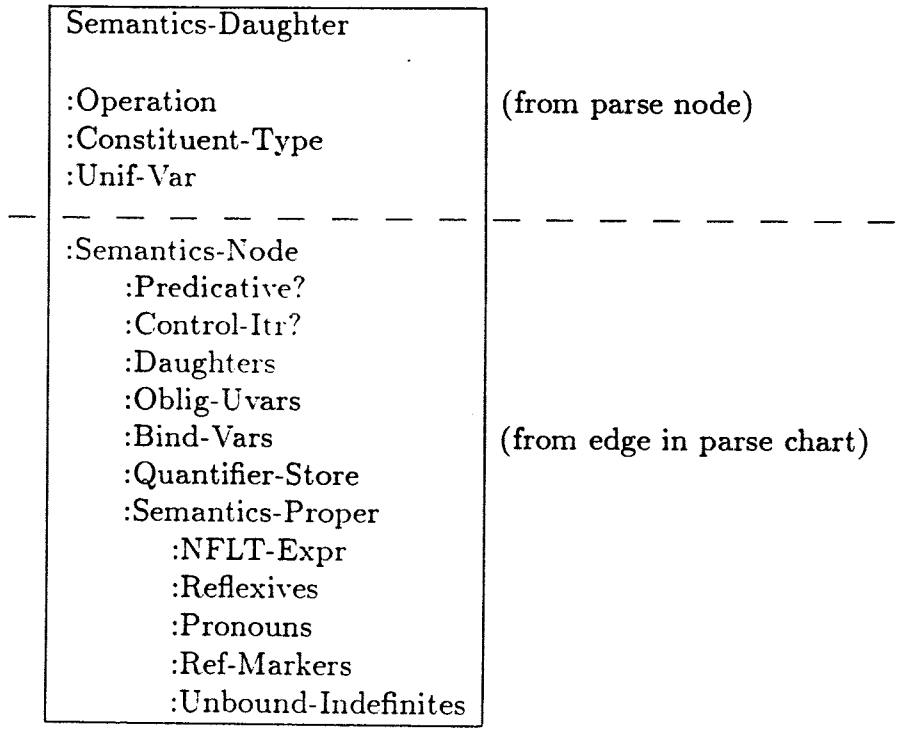
13.2.3 Task of Transducer

The parse chart that is input to the transduction module contains a great deal of information that is irrelevant to semantic processing, so that one task of transduction is to filter. The parse chart also includes a parse tree, which is just a tree of nodes, each of which corresponds to a constituent (phrase, or meaningful unit) of the sentence being analyzed. Each such node contains slots for syntactic information, some of which is also relevant to semantics computation. For example, each node records the syntactic category the phrase belongs to (noun, noun phrase, etc.) as well as optional features (e.g. singular or plural) and further special information indicating whether the phrase is lacking an expected component that will show up later in the tree. As an example of the last sort, the verb phrase node in *What does Jones manage?* will indicate that the expected object of *manage* is missing; this information will be used to determine the function of *what* later in the tree.

Finally, some purely semantic information is included in parse trees, and this must also be included in the semantics tree. For example, parse nodes also include an indication of the procedure required to construct their NFLT translations from the translations of their daughters. The latter slot is normally empty, indicating that a default procedure is appropriate. The parser may have filled information into this slot as it notes that one of the relevant nondefault procedures will be required (this is determined by the syntactic rule the parser used to construct the node).

13.2.4 Contents of Data Structures

The basic semantics data structure is a tree of nodes as shown below:



We examine the components of Semantics-Daughters:

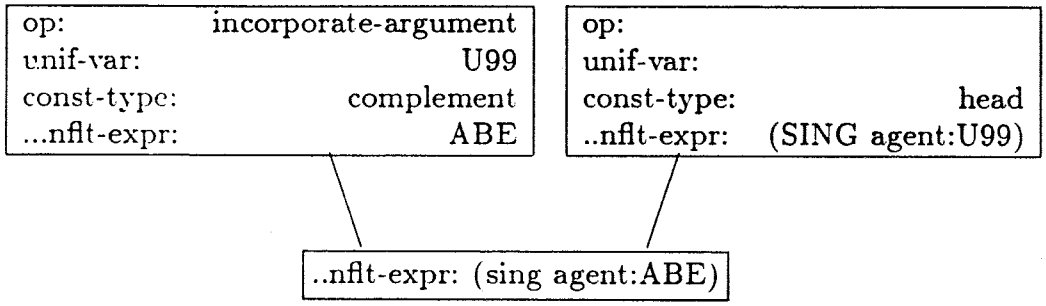
Operation names the semantics operation that will be used to combine the semantics of this node with the rest of the tree.

Constituent-Type is Head, Complement, Adjunct, etc.

Unif-Var (unification variable), if present, coindexes this node with (a part of) the logical translation of the its encompassing constituent, the head. For example, in a parse tree for *Dan works*, *works* might be translated into (WORK agent:U77), while the semantics daughter dominating *Dan* would show *U77* as its *unif-var*.

For example, in the sentence *Abe sang* the node corresponding to *Abe* would specify *Complement* as the constituent type, since *Abe* is a required complement of *sang*; perhaps U99

as the unif-var, where *sang* is translated as (SING agent:U99); and *nil* as the operation, which defaults to *Incorporate Argument*, which will have the effect of unifying the semantics of *Abe* with the semantics of the unif-var U99 modifying the predicate expression as a side-effect, as shown here:



While unif-var, operation and constituent-type specify the function of a node in a larger semantic unit, the semantics node—the last component of the semantics-daughter—contains all the information that isn't dependent on semantic context. Since the semantics is compositional, this includes most of the data.

The semantics node specifies the following information:

P-Edge records the edge in the parse chart from which the information was taken.

Rule-Semantics specifies the (nondefault) rule responsible for combining daughter semantics. Used for constructions other than head-complement or head-adjunct.

Predicative? gets its boolean value from the edges' syntactic feature (PRED).

Control-Itr? gets its boolean value from the edges' syntactic features (SUBJ) AND (SUBCAT).

Oblig-Uvars is used for subcategorizing elements (heads), especially verbs, and lists the unif-vars of its complements; e.g. *sing* has the single obligatory complement indexed by U99 above.

Daughters shows the node's daughters and is used for tree traversal.

Bind-Vars binding variables, keeps a store of variables that will be bound in relative clause and question formation.

Quantifier-Store anticipates a Cooper storage mechanism for the treatment of wide-scope quantifiers (ones that aren't simply left-right scoped).

Semantics-Proper is an object of *Semantic-Data* type that contains storable semantic information. It contains in turn:

NFLT-Expr the logical representation of the node.

Reflexives lists variables introduced by reflexive pronouns that have not yet been bound.

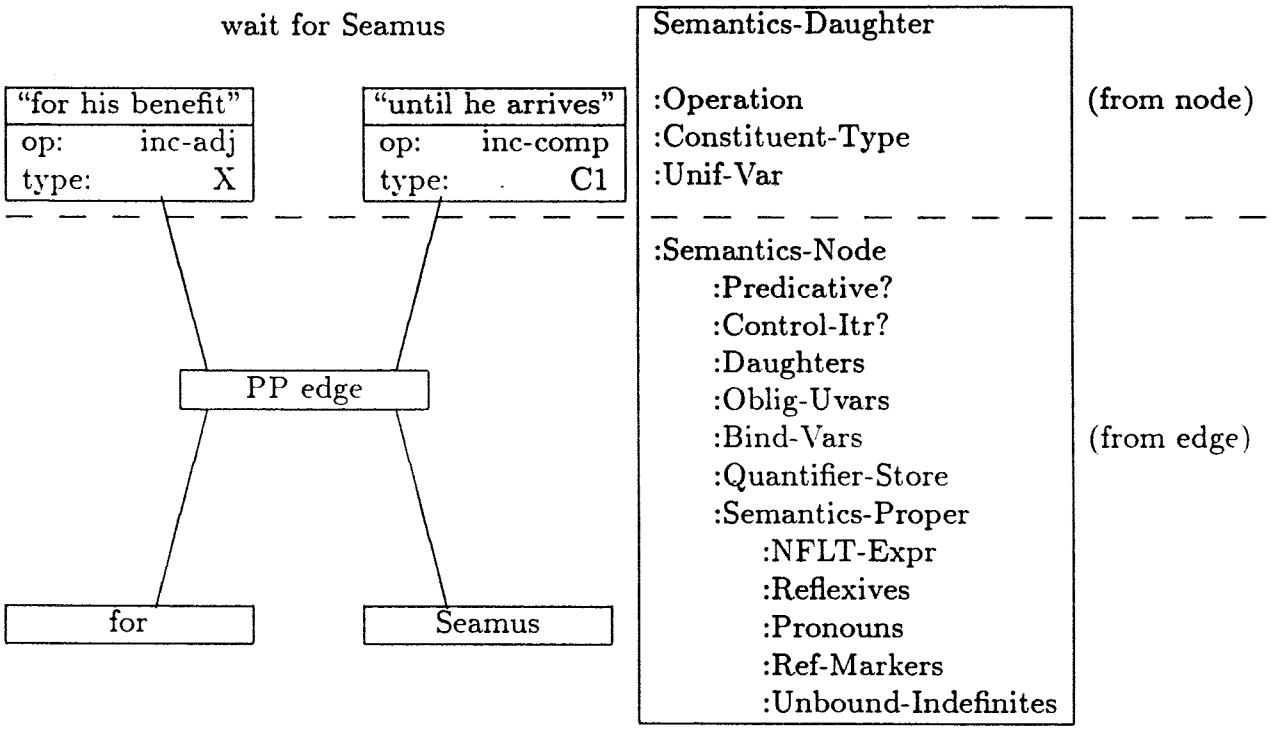
Pronouns works like Reflexives, except for nonreflexive pronouns.

Ref-Markers lists nonanaphoric referring expressions that may trigger contraindexing in principle C. The hope is that this will also serve as a list of potential pronoun antecedents.

Unbound-Indefinites lists unbound indefinite referring expressions, e.g. *a donkey*, whose scope and quantificational force are as yet undetermined.

13.2.5 Daughter/Node (= Parse-Node/Edge) Distinction

A given *semantics-node* contains information from a parser's complete edge (e.g. "daughters" = right hand side of edge; "rule-semantics"; control features; etc.); the data structure *semantics-daughter* contains the information that pertains to the role a constituent plays in a larger structure (e.g. role or unification variable; operation; and constituent-type). To see that this distinction is semantically significant, consider the ambiguity in the prepositional phrase *for Seamus* in:



We draw the line of demarcation in this way for essentially two reasons: compositionality and parsing/semantics cooperation.

On the first, note that the semantic node identifies all the information a constituent contributes to compositional semantics. The semantic daughter, on the other hand, contains information from the context in which the constituent appears. As the example—*for Seamus*—above shows, this may differ for a single phrase. If we did not distinguish semantic node and semantic daughter as above, our adherence to compositionality would be weakened.

Parsing and semantics cooperate better with this division in two ways. First, since the daughter/node distinction mirrors the parse-node/edge distinction, the transduction is simplified. Second, semantics during parsing—"semantics on the fly" or "lockstep semantics"—can operate on semantics nodes; it needn't wait for the context information on semantics daughters. While this is natural given the division here, it would be clumsy without the division (since we would first transduce part of a semantics data structure, compute semantics, then return to fill in further parts).

As far as the mechanics of the system go, the important point is this: the full *semantics-daughter* corresponding to *for Seamus* will contain information not on the edge. The algorithm below for transducing semantics takes this into account by always using the *parse-nodes* to obtain *semantic-daughter* info, and using edges (and their left hand sides) for *semantic-node* info. We keep a back pointer to edges to be able to recognize edges for which semantics has been computed.

13.3 Interface to Parser

13.3.1 Structure Sharing: Design

The parser is careful to conserve space in allowing parses to share edges in the case of lexical and structural ambiguity. The semantics processor must likewise accommodate this sharing by transducing exactly one semantics node for each edge in the parse chart. These are cached (in *Transduced-Edges*) using edges as keys so that duplicate transductions are avoided.

The cache should find further employment (1) in computing semantics during parse time and (2) in treating some types of ellipsis.

13.3.2 Structure Sharing: Current State

The structure-sharing of the parse chart and the structure-sharing implicit in the unification model do not cooperate well. The semantics processor needs the structure-sharing of unification and therefore presently does not accommodate the parser's structure-sharing at all. As far as the semantics processor is concerned, absolutely no structure is shared from one analysis tree to the next. Instead, entirely new structures are transduced. This precludes (effective use of) lockstep semantics computation at the present.

13.3.3 Transduction

This section describes the routines that construct a (sub)tree of semantic nodes. Given a parse node or an initial symbol, the routine constructs a corresponding semantic node, and then calls itself recursively on each of the daughter nodes. The algorithm is essentially the following:

```
(definition transduce-sem-node (edge)
    sem-node  <--- (MAKE-SEMANTICS-NODE)
    for-all syn-var IN sem-node                  ; var. w. syntax info
       sem-node.syn-var <--- edge.lhs.syn-var    ; edge in parse chart
     end-for-all                                 ; lhs - left-hand side
    rhs  <--- edge.RHS                           ; rhs - right-hand side
    if rhs
     then  for-all dtr-node IN rhs
             sem-dtr <--- (MAKE-DTR-NODE)
             sem-dtr.op,unif-var,type <--- dtr-node.op,unif-var,type
             (transduce-sem-node dtr-node.instantiator)
           end-for-all
     else  (fill-in lexical-semantics)
     end-if
    return sem-node
    )
```

The algorithm terminates when there are no daughters in a right-hand side—i.e. at lexical edges. It produces a tree of semantics daughters (as defined above) as output.

There are two types of parser/semantics cooperation we accommodate: (1) semantics postprocessing and (2) lockstep semantics.

```
1. sem-tree <--- (transduce-sem-node  initsym) finis
2. interpolate a first line in transduce-sem-node
     if (semantics-available edge)      ; assume cached semantics nodes
     then  return (semantics-node edge); from above cache
     else    <<as above>>
```

We presently perform all computations after parsing.

13.3.4 Lockstep Semantics Computation

At the same time, we accommodate the computing of semantics in lockstep with the parser. In this regard, note (1) that we can obtain a semantics node from an edge, and (2) that the information from the edge is sufficient to calculate the semantics for any parse node that has it as an instantiator. The hope inspiring lockstep semantics computation is that the parser could use information from the semantics of an edge to make more intelligent choices about which further edges to attempt.

The point of computing semantics in lock-step with parsing is to use semantic and pragmatic information to cut down on parses. Therefore, we can evaluate the control scheme for transducing semantic nodes by asking whether it can abet this process. (Since without the daughter/node division we would transduce from parse nodes, and since this would have to take place later than transduction from edges, it is clear that the use of the division cannot be worse in this respect—parse-node transduction could not possibly guide any more of the parsing than edge transduction.)

Can edge transduction outperform parse-node transduction? To answer this question, we should examine the ways in which semantics might guide the parser. There have been two concrete proposals, one exploiting sortal information, one ontological information.

Sortal information would be used this way. There is more than one parse available for:

a secretary in dcc's phone number

In this example, we'd like to be able to discard the parse where the secretary is inside the telephone number. From the edge dominating the constituent *secretary in dcc's phone-number* we can transduce a semantic node, and this may be used to compute an initial and a disambiguated semantics—which should suffice to discard the edge and avoid edges dominating it. (Note that if we waited to transduce the dominating parse-node, we would construct at least one more edge, so that edge transduction allows a clear savings in work for the parser.)

Ontological information might be employed in a similar way. It is invoked to clarify some

quantifier-scope ambiguities. For example, *each* freely receives wider scope than other quantifiers in the same domain. (Cf. *A representative spoke to each department*, which allows both scope relationships, versus *A representative came from each department*, where *each* is understood as having wider scope than *a*.) The preference for the wide-scope reading of *each* seems to rest on the ontological or metaphysical assumption that someone (or at least a representative) *comes from* a single place. Again, this is information that we could in principle tap at the level of edge, or semantics node.

13.4 Unification-Based Processing

13.4.1 Basic Concepts

Unification has become popular as a computational basis for work in natural language processing, and is introduced in Stuart Shieber's "An Introduction to Unification-Based Approaches to Grammar", *CSLI Lecture Notes Series*, 5.

The basic technique involves regarding the information in linguistic signs as something that is collected as signs are combined. The processing accumulates this information as it recognizes combinations of signs, so that the store of information can only increase during processing. This feature allows clean design.

Crucial to our purposes, unification caters to information that is shared. For example, the agreement information may be expressed in the subject or in the verb phrase (and ultimately its head) or in both. In the "unification-based" paradigm, each of the major components of the sentence is seen as bearing information about the **same** agreement value. The task of processing is to note this information from various sources, and to construct the simplest values consistent with all available information (this is the unification operation).

The semantics processing, while not thoroughly "unification-based" does employ unification to track referential expressions, including placeholding variables, pronouns, reflexive pronouns, the variables bound in quantified expressions, and singular referring expression (proper names), among others. This area was chosen as a target for a first application

of unification-based processing because it involves exactly the information that is often shared among several parts of semantics data structures. Placeholding variables, for example, may figure as parts of the subcategorized-for elements of an expression with a head, its translation into logic (NFLT-Expr), the contraindices found on its store of pronouns, and the controlling index found on obligatory control structures. Unification caters to this shared information.

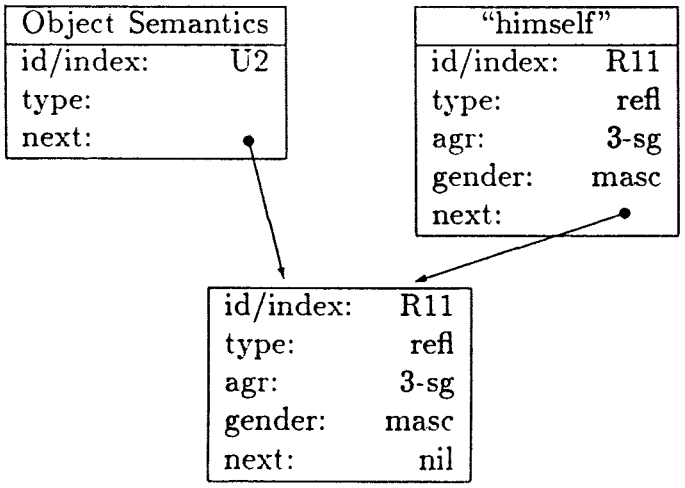
13.4.2 Implementation

Referential expressions are implemented as directed, acyclic graphs. Each such expression is a single node—a degenerate graph—as processing initiates. When two graphs (expressions) unify, pointers within their unique roots are set to point to a common root. The graphs thus combine into a single larger graph, whose unique root is then the (unified) value of the referential expression.

Object Semantics	
id/index:	U2
type:	
next:	nil

"himself"	
id/index:	R11
type:	refl
agr:	3-sg
gender:	masc
next:	nil

These directed graphs combine (unify) to the following graph:



In this illustration, both of the input nodes are destructively modified so that they point to a common root, which will be treated as their effective semantic value.

This implementation of unification is fairly simple, but it has the disadvantage of prohibiting shared structure because it destructively modifies.

13.5 Semantics Computation

13.5.1 The Present Configuration

When the parser hands a parse tree to the semantics module, all the leaf nodes correspond to lexical items (read: words), whose logical semantics (NFLT translations) are given. All the other nodes will specify as yet no translations into NFLT, but they may indicate a nondefault procedure to be used in constructing the NFLT formula from the formulas on its daughter nodes. Semantics computation proceeds bottom-up, going up the tree and filling in the NFLT translations for all the nonterminal nodes in the tree. In many instances this logically involves adding material to NFLT formulas on nodes lower in the tree—effectively destructively modifying the formulas and the nodes. In order to debug the system effectively and in order to share data structures, however, it is necessary to prohibit any actual destructive modification.

When the topmost node in the tree has an NFLT translation, the semantics module is finished, and the resultant formula is handed over to the disambiguator.

13.5.2 An Alternative Configuration

The description above holds for the present configuration, in which semantics processing follows all parsing. In the alternative configuration, in which semantics proceeds in lockstep with parsing, we simply have more frequent calls to the semantics. In this case the parser would call semantics as soon as it successfully constructed a node. Control over these two modes of semantics processing is provided via a global boolean *Local-Semantics*, with

default value NIL.

13.5.3 The Case of Multiple Parses

We avoid destructive modification for debugging purposes: this allows us to see the contents of lower nodes just as they exist at the end of their own semantics computation. But the avoidance of destructive modification is required in any case by the use of structure-sharing.

Since English contains ambiguous expressions, including sentences, the parser may return more than a single parse tree for one expression. In this case the parse trees and the semantics trees share some structures in order to minimize space and processing. For example, this sentence ambiguous:

The managers and consultants working in research attended the meeting.

The ambiguity is in the phrase *managers and consultants working in research.* Should this refer to the consultants working in research and all the managers, or should it refer to just the managers in research (and the consultants in research)? The sentence alone doesn't decide. In this sort of case the parser returns distinct parses for each of the two ways of understanding the phrase *managers and consultants working in research.* The rest of the sentence, *attended the meeting*, is unambiguous, however, and a single parse suffices here. The parser provides two parse trees for the sentence, but each contains a pointer to the node describing the unambiguous verb phrase *attended the meeting.*

Recall now that semantic processing may involve the destructive modification of semantic translations as new information is incorporated into the tree. The examples above illustrate this aspect of semantics as well. In order to incorporate the subject meaning into the sentence, we destructively modify the meaning of the verb phrase *attended the meaning.* It is modified by the addition of the information about who attended. But this means that once we have incorporated a subject meaning, the verb phrase meaning is no longer available to be used in calculating the second way of understanding the sentence.

The semantics module deals with this situation by first creating copies of daughter nodes,

storing these, and then destructively modifying the original. The use of copies solves the difficulty effectively. Since the semantics processor destructively modifies the semantics on parse nodes, and since a single node may be involved in more than one attempt at constructing a semantics translation, some such use of copies is necessary. (At present, we can turn off the copying mechanism by setting the global ***Enable-HPSG-Breaks*** to *nil*. That is, we tie the copying mechanism to the usual development mode, and economically refrain from copying during testing, brief demonstrations, etc. This will not be feasible when we again attempt lockstep semantics computation.)

13.5.4 Algorithmic Control

The method activating the semantics module from the Control Module (see Chapter 8) is the method **process-semantics**, called by the *HPSG-System* method **process**. This transduces a tree of semantics objects and calls the method **compute-initial-semantics** on the tree. This provides a semantics for the sentence nodes.

13.5.5 Bottom-Up Computation

Note that the originating call to semantics is at the sentence level even though the semantic translations are created bottom up. **compute-initial-semantics** calls **compute-semantics**, the workhorse semantics computation routine. **Compute-semantics** works recursively top down, returning values bottom up (and thus constructing the semantics bottom up).

Compute-semantics works on a mother node m in the following way: first, it orders m's daughters from most oblique to least oblique, where the head of the construction counts as most oblique, prepositional phrases as less oblique, direct objects as still less oblique, and subjects as least oblique of all. This is accomplished in **semantically-ordered-daughters**, which relies on syntactic information about obliqueness provided by the parser. Certain particles are allowed to function as semantic heads even though they are syntactically non-heads, e.g. the comparative *than*. If any daughter node has no

associated semantics, there is a call to **compute-daughter-semantics**. Unless the daughter is a trace, this results in a recursive call back to **compute-semantics** to compute the semantics of the daughter's daughters. The calls continue down the tree no further than the leaf nodes, whose semantics are available from the lexicon.

The case in which the daughter is a trace, is handled by **create-binding-dependency**. This will be taken up below.

Once **compute-semantics** has the semantics for an ordered set of daughters, it creates the copies described above, and then checks whether the grammar rule admitting the node prescribed any special treatment (in the parse node slot RULE-OPERATION). If not, the method **compute-default-semantics** is called to create the semantics for the mother node. The default case is the one in which the semantics of a complement or an adjunct is being incorporated into the semantics of a grammatical head. The nondefault cases are enumerated below.

The code discussed here is in **$nlsem/s-node.l**.

13.6 Default Semantics

13.6.1 Grammatical Prerequisites

In order to understand the workings of the default semantics rules, we first review the workings of the parser vis-a-vis the data structures involving *gram-fun*'s (for *grammatical functions*). Intuitively, a *gram-fun* is a specification of a node as it is used by the parser. Thus it includes not just the inherent features of the phrase dominated by the node, but also features derived from its position and function in a sentence. For example, in the verb phrase *hit a man*, the node dominating *a man* is coupled with a *gram-fun* that specifies its inherent features (e.g. (MAJ NOUN)), and a (unif-var) link to its semantic role in this construction, i.e. *patient*. A *gram-fun* always specifies three things: (1) the syntax of the node, a feature bundle; (2) a link to the semantic role that node (more exactly, its meaning) plays (this is the unif-var); and (3) the semantic operation that effects the

incorporation of the node's meaning into that of the head.

A lexical head stipulates its subcategorization requirements through three lists of *gram-fun*'s: Obligatories, Optionals, and Adjuncts. See the grammar documentation for an explanation of the distinctions among these three. The parser, working bottom up, tries to identify a sequence of nodes as the daughters of a mother (for which it then constructs an *edge* in the parse tree). It might, for example, be given the nodes dominating *hit* and *a man*. It first identifies a likely head, in this case *hit*. It then calls the method **subcategorize**, which checks whether the features on the nonhead node may be unified with (the syntactic features on) one of the *gram-fun*'s on the head's subcategorization. In our example, **subcategorize** would check whether *hit*'s Obligatories include a *gram-fun* whose syntax unifies with the features on *a man*. When this condition is met, a pointer to the entire *gram-fun* (including unif-var and semantic operation) is set at the parse tree node for the complement. (**Subcategorize** puts the *gram-fun* into a global register ***Local-Store***, whence **fixup-target** puts it into a Target instance variable on an active EDGE object.)

For our present purpose, it is the result that is most important: lexical heads effectively specify the semantic roles their different complements and adjuncts play; they also specify the required syntax of these. When the parser finds a match between this subcategorized-for syntax and the syntax of a node it has constructed, it hands (a pointer to) the whole bundle of subcategorization information over to the complement or adjunct node. When we come to create a semantics translation for a mother node, we find the information about role and operation not on the head of the construction, where it originated, but on the various nodes with which the subcategorization information has been unified.

This clarified, we turn now to the operation of the method **compute-default-semantics**. This function accomplishes two things: first, it takes the semantics of daughter nodes in the order from most oblique to least oblique and calls, for each daughter semantics, the method **combine-your-semantics-with-head-semantics**; second, it contraindexes pronominal arguments with the less oblique arguments to the same head.

13.6.2 Head-Complement and Head-Adjunct Semantics

We examine the first aspect of **compute-default-semantics** here; pronouns are discussed below. This section examines head-complement and head-adjunct semantic operations. and the following section explains the treatment of control. The function **combine-your-semantics-with-head-semantics** is called to incorporate the semantics of each nonhead daughter node into the semantics of the head. When it has been called for each such nonhead daughter, the semantics of the head includes the semantic information on all the nonhead sisters: this semantics may then be inherited onto the mother node. In general, **combine-your-semantics-with-head-semantics** invokes the semantics operation it finds on the complement or adjunct node (left there in a *gram-fun* by the parser). The semantics operations that may be called this way are listed in the table below, together with examples of the constructions that employ them.

The code discussed above may be found in **$nlsem/s-node.l** and **$nlsem**, unless otherwise noted. **combine-your-semantics-with-head-semantics** calls the following default semantics functions that may be found in **$nlsem/s-dflt.l**.

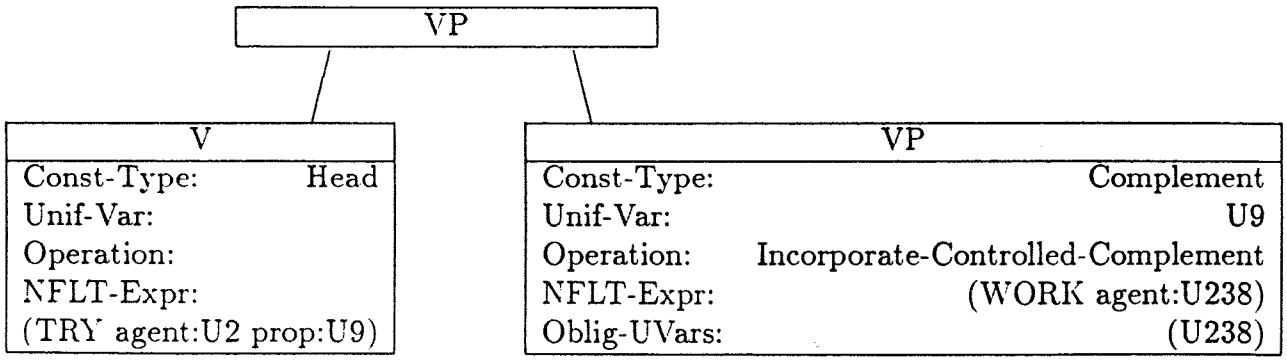
Semantics Operation	Head	Daughter	Example
1. Incorporate-Argument	manage	the NL Project	manages the NL Project
2. Identity-Complement	is	a manager	is a manager
3. Quantify-CN-Phrase	a	manager	A manager
4. Possessify-CN-Phrase	manager	Wasow's	Wasow's manager
5. Incorporate-Controlled-Comp	does	work	Wasow does work
"	try	to work	try to work
6. Assimilate-Loc-Info	work	in NY	work in NY
7. Assimilate-Non-Loc-Info	work	for Jones	work for Jone
8. Incorporate-Head	work	if Al works	work if Al works
9. Full-Relativize	manager	who Dan supervises	manager who Dan supervises
10. Reduced-Relativize	manager	supervised by Creary	manager supervised by Creary
11. Thatless-Relativize	manager	Creary supervises	manager Creary supervises
12. T-Scope-Quantifier	is	a consultant	Is (there) a consultant (in DCC)?
13. Nominally-Modify	manager	department	department manager
14. Incorporate-Title	Wasow	Professor	Professor Wasow
15. Incorporate-Sentential-Complement	if	Al works	If Al works
16. Incorporate-Slashed-VP-Comp	easy	to supervise	easy to supervise
17. Incorporate-Argument-and-Copy	taller	than Bo	taller than Bo
"	taller	than Bo is	taller than Bo is
18. Get-Q-Restrictor	than	oranges	than oranges
19. Comp-Ellipsis3	than	Bo is	than Bo is
20. Subdeletion-Binding	than	Bo is tall	than Bo is tall

13.7 Semantics of Control

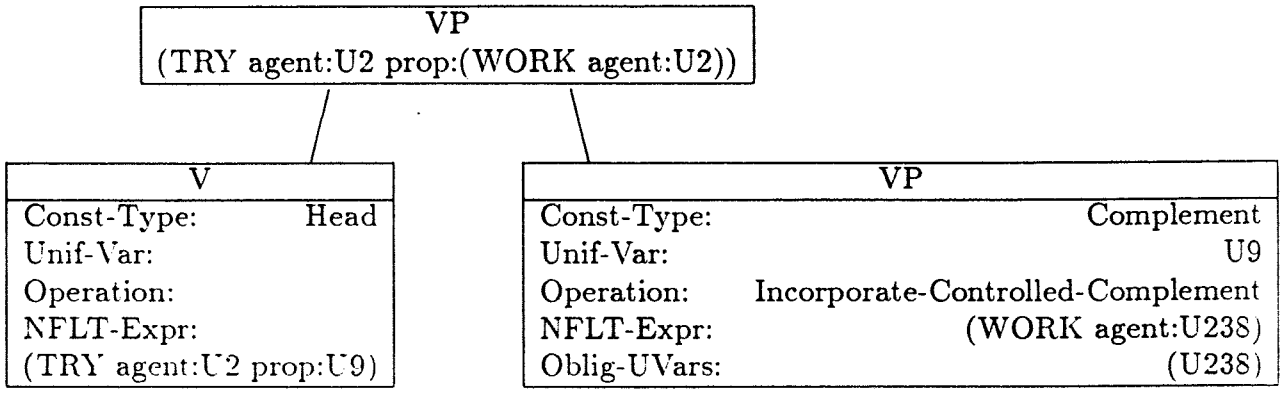
Control phenomena involve "understood" elements. For example, in a sentence such as *Creary tries to work*, *Creary* is the understood subject of work (just as it is the real subject of *tries*). The semantic side of control requires coindexing the agent roles of TRY and WORK, in other words, "understood" subjects fill semantic roles explicitly in the NFLT expressions, so that the translation of the example would be *(TRY agent:Creary proposition:(WORK agent:Creary))*. Since the semantics is "compositional", there is no mention of the eventual subject *Creary* in the semantic for the verb phrase *tries to work*; unification has the effect of first coindexing the subject positions of TRY and WORK, and second unifying the subject position of TRY with its argument *Creary*. (In an alternative formulation of the semantics of these constructions, we may express the agent of WORK in the example above as *self*, which is equivalent to a concept of something in the immediate superformula—i.e. a concept of Creary in this example. Cf. the section on NFLT above for further information on this alternative.)

In order to effect this treatment of control, the controller in the matrix semantics must be unified with the controlled position in the embedded VP's semantics. This is always the controlled complement's subject, and the unification is performed in the **CONTROL** routine called by default semantics.

Beyond the unification of controller and controlled complement's subject, we have simply to incorporate the semantics of the controlled complement into the head semantics. The input to this pair of operations is shown below:



And the result of the bottom-up computation is shown here:



Once the mother acquires this information, it passes it up until an argument is encountered, in this case the subject. When the subject meaning is incorporated into the semantics, its meaning is unified with U2, effecting the desired semantics output: (TRY agent:U2 prop:(WORK agent:U2)).

13.8 Rule Semantics

Several of the sixteen grammar rules currently used in the HP-NL system are marked as requiring special semantics treatment. Each of these deviates from the default case, in which the semantics of a complement or an adjunct is incorporated into the semantics of a grammatical head.

Five of the rules, their semantic interpretation, and examples of their application follow. The other two rules involve coordination, whose semantics is taken up in the following section.

Semantics	Grammar Rule	Example
Topicalize	TOPICALIZED-S-RULE	Who + does Dan supervise? who + Dan supervises (relative clause) That guy, no one can supervise.
Existentially-Quantify	BARE-PLURAL-RULE	Managers (met). $manager(x) \rightarrow (\exists x) manager(x)$
Get-Semantics (Comp)	PP-NONPRD-RULE	(rely) on + Jones (on'+Jones'=Jones')
Get-Semantics (Head)	POSSESSIVE-NP-RULE	Jones + 's (Jones'+('s)'=Jones')
Quantify-Type-Lowered-Quantifier	PARTITIVE-RULE	one + of the men = one man

The relevant code is found in **$nlsem/s-rule.l**.

13.9 Natural Language Constructions

13.9.1 Coordination

Sentential coordination (*It's raining and it's cold*) presents no special difficulty to the semantics processor: they are simply mapped onto conjunctions (and of course disjunctions, which we henceforth ignore) in the logical language NFLT. But every natural language allows coordination of a wide range of syntactic and semantic categories, most of which are equivalent to sentence conjunctions without subsentential coordinations. The processing problem boils then down to this: how do we map subsentential coordination onto sentential conjunction?

The strategy that's been adopted is to interpret coordinations as NFLT coord-exprs. In the sentence:

Tom, Dick and Harry work

Tom, Dick and Harry is translated as the coord-expr:

```
(AND Tom, Dick, Harry) {AND (Coord-expr)}
```

(The AND stands for coordination, where no decision has yet been made as to whether a distributed or collective reading is intended.)

A coord-expr is later incorporated into a predicate (or sentence) expression, e.g (WORK agent:U5), just as a simple argument would be, yielding (WORK agent:(AND Tom, Dick, Harry)). At the discourse level, the decision is made whether to interpret the coordination collectively or distributively. If it is interpreted collectively, it is in its final form. If it interpreted distributively, then the predication is distributed over the parts of the coordination by the **distribute-coord** method, resulting in a sentential conjunction form:

```
(AND jnct:(WORK agent:Tom) jnct:(WORK agent:Dick) jnct:(WORK agent:Harry))
```

Our present fragment includes no predicates with nondistributive readings. The discussion above represents the present processing of the distributive readings of subsentential coordinations and a partial design for the treatment of collectives.

The relevant code may be found in **$nlsem/s-rule.l**.

13.9.2 Unbounded Dependencies

An unbounded dependency is a dependency that can hold between elements separated by any number of clause boundaries. The HP-NL system treats three of these: questions, relative clause formation, and topicalization. Examples of these follow:

Who does Creary supervise?
The manager who Creary supervises works in DCC.
That researcher, Creary supervises.

In each of these cases a dependency exists between a displaced element and the position from which it is displaced. The HP-NL treatment is designed to scale up gracefully as we incorporate new constructions that would allow the length of dependency to be increased.

The parser recognizes a construction from which expected material is missing; e.g. it expects to find a grammatical object with the verb *supervise*. When no grammatical object is found locally, the parser does the following: (1) it introduces a "trace" into the parse tree that stands for the (locally) missing element, and it interprets this trace through a variable (thus the variable is listed as the semantics of the trace); (2) it introduces a value to the feature [SLASH] corresponding to the category of the missing, but expected element; (3) it assigns the meaning of the newly introduced trace to the feature [BINDING-VARIABLES] (referred to as [BIND-VARS]). The second of the parser's actions is motivated syntactically and won't be discussed further; we continue with the others.

Once the trace has been introduced into the tree and interpreted semantically, we can manipulate it using all of the same semantic rules we used for explicit phrases. To continue the example above, we may use the operation **incorporate-argument** to obtain a semantic translation for the phrase *supervise* ⟨ *trace* ⟩, etc. This is the semantic justification for the use of the trace and its variable meaning.

We turn then to the use of the feature [BINDING-VARIABLES]. A binding variable is passed from daughter to mother up the parse tree until it is reset explicitly by the Topicalized-S-Rule, whose workings we consider presently. This feature is motivated semantically: it indicates the presence of a dependency one of whose elements has yet to be encountered by the semantics processor, and it provides a way of linking the semantics of the missing element with the context from which it came.

We leave the description of processing at the site of the trace, and we now examine the processing of the displaced material, or "filler". This is the question word *who*, the relative pronoun *who*, and the noun phrase *that researcher*, respectively, in the three examples above. We take the case of questions; the others are similar.

The question word *who* is treated in a way parallel to the trace. (1) it is assigned a variable as preliminary semantics so that further semantic operations may apply normally (i.e. we ask not only *Who...?*, but also *Whose manager...?*, *Which manager in DCC who has a secretary...?* etc. These complex fillers are analyzed using standard rules. (2) the question word is assigned a syntactic feature indicating its status, [QUE], which abets proper syntactic treatment. (3) it too uses [BIND-VARS] to maintain the information about the variable assigned as meaning to the question word. This variable (and some additional information) then appears as the value of the [BIND-VARS] feature, which is normally passed from daughter to mother up the tree.

The two halves of the construction must be joined at some point. Syntactically, the point of juncture is simply the first encompassing node, i.e. the node at which *who* is joined to *does Creary supervise*, or at which the relative pronoun is joined to the remainder of the relative clause, or the topicalized element to the remainder of its topicalized clause. Note that each of these combinations is licensed by the same syntactic rule, TOPICALIZED-

S-RULE. We refer to the two daughters admitted by the rule as the filler (*who*) and the gapped clause (*does Creary supervise*).

Semantically, however, we combine information in two steps. The first step is the operation **topicalize**, which is called by the syntactic rule TOPICALIZED-S-RULE. **Topicalize** unifies the binding variables on the two daughters. In the case at hand, *who*'s semantics is unified with the semantics of the binding variable on *does Creary supervise*.

Here's an example:

```
'who'
  SEMANTICS   = "B8"
  BIND-VARS   = ((QUE [B8, {WHICH P88 (PERSON inst:B8)
                                      [Not yet scoped.] }]))
```

combines with:

```
'does Creary supervise'
  SEMANTICS   = (MANAGE agt:CREARY ptnt:B23)
  BIND-VARS   = ((SLASH B23))
```

to obtain:

```
'who does Creary supervise?' at S node
  SEMANTICS   = (MANAGE agt:CREARY ptnt:B8)
  BIND-VARS   = ((QUE [B8, {WHICH P86 (PERSON inst:B8)
                                      [Not yet scoped.] }])
                 (SLASH))
```

Of course, this still isn't finished—there isn't a well-formed NFLT formula assigned as semantics. The final step is accomplished in a special routine **que-retrieval** called by **compute-semantics** at the S node. Essentially, this sets the semantics computed up to that point to be the scope of the quantifier in BIND-VARS:

```
'who does Creary supervise?' at later S node
  SEMANTICS   = {WHICH P92 (PERSON inst:P92)
                           (MANAGE agt:CREARY ptnt:P92)}
```

The semantic processing of topicalized sentences is very much like that of questions; it differs in calling not **que-retrieval**, but only **topicalize**. We include an example of the semantic processing of relative clauses, however, in order to clarify one design decision. Let us examine the data structures at the point of combination:

```
'who'
  SEMANTICS   = ''B3''
  BIND-VARS   = ((REL [B3, (PERSON inst:B3)]))

'Creary supervises'
  SEMANTICS   = (MANAGE agt:CREARY ptnt:B25)
  BIND-VARS   = ((SLASH B25))

'who Creary supervises'
  SEMANTICS   = (MANAGE agt:CREARY ptnt:B3)
  BIND-VARS   = ((REL [B3, (PERSON inst:B3)]) (SLASH))

'manager who Creary supervises'
  SEMANTICS   = (MANAGER INST-BIND::P98
                    s/t:(AND jct1:(PERSON inst:P98)
                             jct2:(MANAGE agt:CREARY ptnt:P98)))
  BIND-VARS   = ((SLASH) (REL))
```

In the question example, we might have derived the final semantics in a single step rather than in two. In the relative clause construction, however, further binding (to the argument position of the head noun) is required. For this reason, the semantic operation **topicalize** doesn't itself perform the binding.

13.9.3 Comparatives

Natural language comparison implicitly relies on the existence of comparable ratio scales for its functioning. The scales are presupposed even with respect to properties for which we can point to no reasonable measure, e.g. *beauty*:

Kim is { *much* / *somewhat* / *a lot* / *enormously* / ⋮ } *more beautiful than Sandy*

Kim is twice as beautiful as Sandy

This being the case, our semantics for comparatives employs scales along which individuals may be compared. For example, the following represents the semantics of *Lyn is taller than 5 ft.*:

$$\exists q(q > 0 \ \wedge \ (\_\_\text{exceeds}\_\_\text{by-at-least}\_\_arg1 : (height(Lyn))arg2 : (5ft)arg3 : q)$$

That is, Lyn's height exceeds five feet by some positive quantity. The implicit scale is that of height, and the differences between points on the scale is assumed to be significant.

Given the existence of scales along which properties may be measured, we also obtain a semantics for uncompared adjectives. The semantics of *Lyn is tall* represents Lyn's height as at or above a default standard for tallness:

$$\exists q(q > 0 \ \wedge \ (\_\_\text{exceeds}\_\_\text{by-at-least}\_\_arg1 : (height(Lyn))arg2 : (dflt - height)arg3 : q)$$

Thus the semantic distinction between the comparative and positive forms of the adjective is that the comparative fills in an argument position ($arg2$) with material found in the sentence, while the positive binds the same argument position to a default standard.

The use of a basic three-place predicate as the translation of simple comparable adjectives leaves the semantics proper some flexibility: we can represent the notion *exactly n feet tall* as well as *at least n feet tall.* In fact, we exploit this flexibility and render *Lyn is n feet tall* as "Lyn's height exceeds n feet by some positive amount". Thus, if Lyn is n feet tall, it will follow that she is n − m feet tall, for m < n. It surely normally follows from any assertion of *Lyn is n feet tall* that she is therefore *exactly* n feet tall, but this may be seen as a consequence of the conversational maxim of quantity: say as much as is true. This

analysis is currently implemented. Nothing in the semantic apparatus forces this decision, but neither is there any difficulty in accommodating it.

13.9.4 Anaphora

"Anaphora" refers to pronominal reference, and there are two classes of pronouns, nonreflexive: *he, she, it, they, him,..* and reflexive: *himself, etc.* We treat both, but this section is devoted to the treatment of nonreflexive pronouns.

There is an important division of labor between the semantics processor and the pragmatics processor in the treatment of pronouns. The semantics processor is responsible for cataloguing the presence of pronouns within a sentence, for representing pronominal reference within logical formulas, and for noting constraints on the pronoun-antecedent relation that arise due to grammatical circumstances. The pragmatics processor, on the other hand, is responsible for interpreting the reference of pronouns within the context of a discourse. Since every pronoun, including those with likely intrasentential antecedents, might be interpreted as discourse-bound, this division of labor is designed to affect the processing of every pronoun.

Pronouns are represented as a (unifiable) type of variable within semantics data structures, and they function as singular terms (e.g. proper names) within logical formula. All the pronouns within a sentence are collected into a **Pronouns** store that is then handed to the pragmatics processor, which has the task of finding optimal antecedents.

To conduct this search, the pragmatics processor requires that pronouns bear all the information relevant to constraining the search for their antecedents. This includes person, number and gender, which is lexically provided (and semantically transmitted), as well as contraindexing information, which depends on grammatical circumstances, but whose calculation is performed within the semantics processor.

Pronoun Contraindexing

Two types of contraindexing information are incorporated: one based on the subcategorization domain and one based on the command relation within an analysis tree. In both cases, we contraindex a pronoun with other referring expressions and interpret this as a presumption against their coreference. For example, if we contraindex an instance of *him* with *Jones*, we constrain the search for *him*'s antecedent so that *Jones* is strongly dispreferred as antecedent.

Subcategorization contraindexing takes place within the clause or noun phrase, the maximal domain of subcategorization:

Tom manages him.
Tom described Bill to him.

Subcategorization contraindexing marks the pronoun *him* as contraindexed with each of the noun phrase in the sentences above. It thus functions like "Principle B" of the binding theory developed in Chomsky's Government and Binding Theory. The generalization employed in HPSG: a pronoun that fills a subcategorized-for slot of a given head is contraindexed with each of the less oblique elements on that head's subcategorization list. (This might be made more general, so that a pronoun is contraindexed with the more oblique elements as well, but it turns out that command contraindexing covers these cases in English.)

Since nouns also subcategorize for complements, subcategorization contraindexing also accounts for the noncoreference of *Tom* and *him* in the following sentence:

Dan read Tom's report about him.

It allows that *Dan* may be the pronoun's antecedent both here and below:

Dan read the report about him.

Command Contraindexing, on the other hand, operates between a pronoun and the nonpronominal noun phrases dominated by its sisters in the analysis tree. Here are some

examples of pronouns in command contraindexing relationships:

He read the report about Dan.
He knew that Dan would work with Tom.
She persuaded him that Dan had read the report about Tom.

Note first that each of the pronouns in the example sentences commands all of the nonpronominal noun phrases in its sentence. One may see this by applying the definition of "command" given above. The pronoun subjects are sisters to their respective verb phrases, and they therefore command the pronouns appearing in the verb phrases. The pronoun object of "persuaded" is in a similar relationship to the noun phrases in the sentential complement.

We next verify that the pronouns are indeed presumed to be noncoreferential with the noun phrases they command, and then consider that none of these cases of contraindexing could easily be brought under the rubric of subcategorization contraindexing. This confirms the need for a distinct type of contraindexing along the lines of the Government and Binding Theory's "Principle C".

Two versions of the principle are supported in the code: the default version generalizes the principle and contraindexes pronouns with all nonpronominal noun phrases that follow it, even those which are not in a command relation proper. Thus it contraindexes the pronoun and nonpronominal noun phrases below:

A manager supervising him collaborates with Dan.

The ability to recognize *Dan* as the antecedent seems superfluous in the style of dialogue we've examined. Still, there is a nondefault version of command contraindexing in which this is possible. In this version, pronouns are contraindexed only with the nonpronominal noun phrases they properly command. We've also experimented with various definitions of the "command" relation: the main distinction among these is whether the definition depends only on the configuration of the analysis tree (this is the version presented implemented and implicit in the definition above). Alternatively, the state of subcategorization may figure in the definition of command.

Indefinite Antecedents (Donkey Pronouns)

There are two sorts of noun phrases: those with true generalized quantifier interpretations, such as *Every manager* or *Most employees*, and those with indefinite interpretations, such as *An engineer* or *Some secretary.* The latter type has the ability to function as the antecedent to pronouns that lie outside of their naturally induced scope. These have become known as "donkey pronouns" because they were first noted in examples of the following sort:

Every farmer who owns a donkey beats it.

These noun phrases present two challenges: first, we need a definition of their scope that allows that a "donkey" pronouns such as the one above be determined to lie within the scope of its (possible) antecedent; second, and evident in the same example, we need a prediction of the logical force of such noun phrases. We see this from the most likely rendering of the sentence above in predicate logic, i.e. one in which the indefinite determiner is translated as a universal, not an existential, quantifier.

There is a response to both of these challenges in the present code:

In order to use indefinites outside their proper scope, we keep track of which reference markers are introduced by indefinite determiners, and we allow pronouns to employ these "ref markers" as antecedents; in case the scope of the original indefinite would not have included the pronoun, appropriate conditions are placed on it so that truth conditions are preserved. Both references are ultimately rendered by existential quantifiers:

A man entered. $\exists x(man(x) \wedge enter(x))$

He sat down. $\exists x(man(x) \wedge enter(x) \wedge sit(x))$

In order to determine the proper logical force of indefinite noun phrases, we check for "trapped" indefinites whenever real quantifiers (or conditionals) are introduced. This triggers a reinterpretation of the indefinite as a universal quantifer in case the indefinites are trapped within the antecedent of a conditional, and an appropriately restricted existential in case the trapping environment is a quantifier:

If a man entered, he sat down	$\forall x((man(x) \wedge enter(x)) \leftrightarrow sit(x))$
Every farmer who owns a donkey beats it	$\forall f(\exists d(farmer(f) \wedge donkey(d) \wedge own(f,d)) \leftrightarrow \exists d'(donkey(d') \wedge own(f,d') \wedge beat(f,d'))$

Thus the classic "donkey sentence" is rendered: every farmer who owns a donkey beats at least one of the donkeys he owns. Simpler renderings would be possible, but they seem to make no sense of the nonequivalence of pairs such as the following:

Most farmers who own a donkey beat it.
Most donkeys owned by a farmer are beaten by him.

(The processing of indefinites and anaphora is now under revision, and some aspects of the treatment sketched here, including at least the treatment of indefinites trapped by quantifiers, are expected to change.)

13.9.5 Reflexives

Reflexive pronouns are understood as coreferential with the "subject" of the construction they appear in.

Tom manages himself.
Tom read the report about himself.
Dan read Tom's report about himself.

In each case *himself* is understood as coreferential with *Tom*, which is clearly the subject of the first two sentences, and which might be regarded as the (possessive) subject of the NP in the last sentence.

When a reflexive is encountered in bottom-up processing, we represent its meaning using a variable R1, R2, etc. This is manipulated in the same way that a proper name might be. For example, we represent *manages himself* at one level as:

```
(MANAGE patient:R2)
```

At the same time, we store the variable R2 in the Reflexives store, indicating that further processing is required higher in the tree.

The treatment of reflexives turns crucially on the recognition of words such as *manage* or *picture* that may have subjects. It turns out that these may be characterized as the words (more generally, words or phrases) that assign *gram-fun*'s to other phrases. This is equivalent to saying that they subcategorize for the other phrases. The subject is then the last item they subcategorize for—i.e. the last to which they assign a *gram-fun*. We recognize subcategorizing elements semantically by checking (1) the CONTROL-ITR? value on the semantics node, and (2) by checking whether the last *gram-fun* that gets assigned really is a "subject". This information is found on the Oblig-UVars slot on semantics nodes.

Once we reach the subcategorizing element, we remove the reflexive variable from the Reflexives store and unify it with the subject's semantics, represented by its unification variable.

The manipulation of reflexives is done by the Reflexive-Handler in **$nlsem/s-node.l**.

13.10 User Interface

The most important tools for semantics viewing are the softkeys accessed under Semantic Vars (in the HPSG softkeys), and the information available under the softkey Examine Node. The softkeys accessed under Semantic Vars are Initial Semantics, Discourse Semantics, Disambiguated Semantics, and Database Query, which provide the final representations at those levels (once a sentence has been processed), and Set IS-list, Set DCS-list, Set DS-list, and Set DQ-list, which set their variables to Initial Semantics, Discourse Semantics, Disambiguated Semantics and Query, respectively. The use of the Examine Node softkey allows one to examine the semantics of interior nodes as well as the important *gram-fun* data structures.

For semantics hacking, the basic NFLT definitions and methods described in Section 2

above and coded in **$nlsem/nflt.l** are most important.