1. INTRODUCTION

Contents:

This document is designed to give the reader an in description of the natural language generation system Mumble-86. We have attempted to include sufficient detail that one can not only understand the system, but be able to use it as a component in a larger system.

In this introduction, we look at Mumble's place in the generation process and the design principles that have guided our work. We also present a historical overview of the evolution of Mumble from McDonald's early work in generation at the MIT AI Lab, its culmination in his thesis in 1980, and the changes that have led us to retitle the current system Mumble-86. We pay particular attention to why we have removed much of the responsibility for selection and planning that have been present in earlier versions.

Section Two is a detailed summary of Mumble's design, including its levels of representation, major processes, and representation of the grammar. In Section Three, we take a step closer to the implementation. We look at the object types and algorithms in the virtual machine and at the supporting sub-systems for defining types and for tracing and displaying the state of the processes. In Section Four we present a detailed walk through of Mumble generating a sentence.

Section Five provides information about the actual program: the directory structure, how it loads, and how to install it on your own host. In Section Six, Interfacing to Mumble, we look at how you build an interface between an underlying application program and Mumble. We show examples from some of our current projects. Section Seven gives a detailed account of how to use our interactive demo facility, which guides the user in building new input specifications for Mumble.

1.1 The components of the generation process

The question of what goes into a natural language generation system or "generator" or of where one draws the boundaries between its components will be answered differently depending upon who one talks to. For the purposes of this document we will adopt the following rough breakdown into three top level components or processes:

Underlying program -- Developed independently of the generator per se, this will be the expert diagnostician, cooperative database, ICAI tutor, etc. that the

human users want to talk with. Some event within this underlying program will initiate the generation process and the determination of the goals the utterances are to achieve is its responsibility.

Planning -- This process determines how the goals can be achieved in a given context. This includes selecting the information to be communicated (or omitted), determining what perspectives and rhetorical organization the information should be given, and choosing a mapping for the information onto the linguistic resources that the language provides (i.e. open-class words and syntactic constructions).

Realization -- This process carries out the planner's specifications to produce an actual text. It has the responsibility for insuring that the text is grammatical, and will handle the bulk if not all of the syntactic and morphological decision making.

In these terms, Mumble-86 is a realization component. We also refer to it as a "linguistic component", reflecting the fact that all of the planners and underlying programs that Mumble has been used with to date have concentrated on conceptual issues and left all of the linguistic efforts to Mumble; this designation may have to change in the coming years as the semantic and discourse level contributions of earlier components become more significant.

1.2 What Mumble-86 expects as input

We expect any system that uses Mumble-86 as its linguistic realization component to be able to supply the following kinds of information about each utterance that it wants produced, using the specification language that we supply (see Section 2.2).

(a) The units from which the utterance is to be composed. The mapping for each unit to its intended linguistic resource will either have been already made or will be fully defined for execution after realization has begun.

(b) The functional relationships among the units, e.g. predication, head, modifier, given, theme, etc., that direct or constrain their organization within the text.

(c) Lexical choice. As the primary means of delimiting what information is or is not communicated and what perspectives and connotations are presented, all open class words are choosen by the planner.

Given that information, the following work must be done to then produce a text. This work is done by Mumble-86:

(a) Assembly of the text from the specified pieces in a way that guarentees it will be grammatical and express the indicated functional relationships.

(b) Maintaining all needed syntactic context and morphological specializations.

(c) Under the guidance of that structure, defining and applying contextual constraints on realizations, e.g. the difference between "*little*" as a modifier and "*is little*" as a predication.

We have choosen to draw the line here principally because we think that realization (as we have construed it) is a well established and clearly delimited component, possibly the only such component in the generation process today, given the state of the research. As such, we feel comfortable in distributing it to other groups in generation and related research areas without believing that we have unduely burdened them with presumptions about the kind of representations to be used in their underlying programs or with how planning is to be carried out.

1.3 Our approach to generation

In our approach to generation, we maintain that the way the decisions are organized, who is in control of the decision making, and the structure of the reference knowledge used in the decision making have an impact and the efficiency and effectiveness of the generation system. We can summarize our approach in the following design principles:

--Modular design with divisions based on the kind of information brought to bear and the type of representation used.

--Multiple levels of representation with each level adding constraints and providing context for further decisions.

--Active representations where the mapping from one level to the next is controlled by the higher level rather than by static reference knowledge.

--Minimal units of choices carefully chosen to reflect a natural granularity.

--Predefined choice sets defining the set of possible choices annotated by the context each may be used in.

We will not go into detail describing and justifying these principles (see McDonald, Meteer, and Pustejovsky, 1987), however if you keep them in mind as you read the details of our design you will see their effect. For example, our concern form modularity is reflected in the division of the generation process into "planning" and "realization" described in section 1.2. Multiple levels of representation is reflected in the three explicit, active levels within Mumble: the realization specifications, the surface structure, and the word stream. Finally, we have chosen the phrase as a significant minimal unit and have predefined sets of phrases as the choice sets.

1.4 Historical background

The name "Mumble" has been used with a series of natural language generation systems starting in 1972 with McDonald's reworkings of the generator in Winograd's Shrdlu. This can lead to confusion since the only real constant across the different versions has been the continuity of the primary author, while the scope, competence, and internal design have undergone immense change. In retrospect it is clear that the better thing would have been to follow the usual practice and rename the system with each significant revision, even though at the time the changes always seemed incremental and evolutionary. This section will sketch the versions that Mumble has gone through, associating them with the appropriate references and pointing out their major characteristics.

There have been two major trends in this research. One is the gradual evolution from an implementation based purely on "raw" Lisp code, with the discipline only implicit in the author's mind, to one built on a precisely defined virtual machine and a set of very-high-level, task-specific languages. The first full implementation of Mumble in 1975 consisted only of functions and global variables; Mumble-86, on the other hand, is fully schematized and does not permit a user to use any Lisp code at all.

The second trend is an ever increasing differentiation of representation and mechanism within the total generation process. Mumble in 1974 was responsible for the entire generation process including the initiating urge to speak; Mumble-86 is intended as a narrow "realization" component only, with all of the planning and specification of goals already established by other components acting in coordination. The motivation here is in part just conventional design wisdom: decreasing the number of implementation options for handling new phenomena will simplify the programmer's effort. Its other aspect is metatheoretical, namely to increase the viability of the design as a candidate psychological model by reducing the power and scope of the individual processes and representations.

The original concept paper on Mumble, McDonald 1974, viewed generation as divided into two interleaved processes of "composing" (comparable to today's "text planning") and "realization". It included the notions of an explicit message distinct from any data structures already formed in the underlying "reasoning" program, of surface structure as an intermediate representation, and of incremental, left-to-right production.

The first implementation took place in 1975, using ad-hoc single examples drawn from various reasoning systems that were being developed in the MIT A.I. Lab at the same time. Over sucessive years it underwent piecewise improvements as parts of the process became sufficiently well understood to be replaced with specifically designed schematic languages and their interpreters. Interesting papers from this period include McDonald 1975, 1978a, and 1978b.

This period of Mumble's development culminated a paragraph-length, linguistically sophisticated example of a fluently expressed natural deduction proof (the "barber" paradox) represented in the predicate calculus. (This is the example written up in McDonald 1983, though the work was done in 1977 and in part appears in 1978a.) At this stage in the design an already existing data structure (e.g. the proof) would be passed directly to Mumble as its input, and a custom set of "dictionary entries" would recursively work their way through the structure deciding what phrasings to use in realizing its elements.

After this milestone, a major part of the work was the clarification of the interface between the representational system of the underlying program and the dictionary entries that had been developed for it. This led to interfaces to the representations KL-ONE, OWL, FRL, the predicate calculus, and simple semantic networks. The first consideration of how Mumble might be applied as a psycholinguistic model of language production also occurred in this period; many of the ideas were ultimately published in McDonald 1984.

McDonald's Ph.D. thesis (1980, unpublished) was the formalization of this "1979" version of Mumble as a set of abstract data types (the present "type" system, see section 3.2.1), along with a presentation of the syntactic analyses used and the rationales behind their formulation. The 1983 paper is a good summary of the philosophy of generator design that was espoused in that period. The key element is the centrality of the surface structure of the utterance being produced: it is simultaneously the output representation of the realization process and the control structure directing that process' overall operation.

After 1980 the program was completely rewritten to reflect the new, schematic design. The next significant event was Jeff Conklin's Ph.D. work (Conklin, 1983), in which he developed the text planner "Genaro" for the task of describing picture of houses as analyzed by the UMass Visions system. Conklin used a simple threshold-based algorithm to select objects and relations for inclusion in the text, reacting to the relative salience of the items in the picture (Conklin et al., 1983, McDonald & Conklin 1982). Genaro was the first independent text planning system to be combined with Mumble, and it occasioned a general re-thinking of the proper place of a "linguistic component" like Mumble within the generation process.

During this period the idea of powerful, customized "dictionary entries" was abandoned as unable to capture the generalizations that were becoming apparent (cf. McDonald 1981). The entries had amounted to a piecemeal text planning system embedded as part of Mumble, and it became clear that such efforts were better organized in terms of their own level(s) and representation(s), rather than forced into the surface structure directed control regime of Mumble. Their niche in the design was taken over in 1983 by "realization classes" (section 2.3.1), which are a much more restricted and linguistically oriented device.

This change started the present move towards a far more restricted and differentiated design. Narrowing the kind of information that could influence a decision within Mumble has had the beneficial effect of making the system easier to understand and to use; at the same time, it has ment that Mumble per se -- the code that we distribute under that name -- has much less competence within its scope than it once had: We could not, for example, replicate the production of the "barber proof" (McDonald, 1978a) by just passing lines of predicate calculus directly as input to Mumble; instead, today there would have to be a text planner developed to do much of the work that had formerly been Mumble's responsibility (e.g. choice of wording, organization as a discourse, judgements of what information to include or leave out).

Building on the new view of how a grammar was organized that the realization classes provided, McDonald and Pustejovsky developed an initial theory of the "attachment" process during the course of 1984. (See section 2.6.) The notion of attachment arose from very early work with Mumble on viewing some raising verbs as functionally equivalent to hedging adverbs (e.g. "seems", "almost"), and from our exposure to Tree Adjoining Grammar (Joshi 1983, 1987). The issues and our treatment are discussed in McDonald & Pustejovsky 1985b and 1985c.

The addition of attachment to Mumble removed an earlier requirement that the structure of the final text precisely mirror the compositional structure of the input message. Given this flexibility, we proceeded to explore the possibilities of capturing some of the surface conventions of prose style by controlling what attachment alternatives were choosen as a text was assembled (McDonald & Pustejovsky 1985a). It was possible to define sets of a priori preferences (e.g. prefering to add a relative clause rather than start a new sentence) that were sensitive to a surface description of the utterance's form so far. Ultimately, however, we were dissatisfied with the complexity of the design and its lack of any deeper criteria for text form, and we eliminated the "style" facility from Mumble when we made the revisions that led to Mumble-86 and removed the "history-keeping" facilities that were at that point not being put to any other purpose.

The version of Mumble of 1984/85 was distributed to several sites, notably the University of Pennsylvana and Stuttgart University. At Penn it was integrated into a number of ongoing knowledge-based system projects, and was used as an alternative to the original realization component in McKeown's TEXT system (Rubinoff 1986).

By the fall of 1985 it had become apparent that we knew enough about what kinds of actions linguistic realization really needed to consider dropping from Mumble any ability to use arbitrary Lisp code directly in the rules -- a capacity that had been central to the earlier, "1980", design as a natural part of its data-directed control structure. Our new design of course kept the data-directed control structure (a central tenant of our whole approach), but now we were prepared to provide an exhaustive list of the operations that the grammar writer could have available, in

effect the instruction set of a virtual machine. This new design became the "Mumble-86" that this paper documents.

Between the fall of 1985 and the summer of 1987 the code for Mumble was entirely rewritten from the ground up by Marie Meteer, David McDonald, and Scott Anderson. In the course of this work a careful examination was made of all of the computational devices within the system and of how they were intended to be used. A great many simplifications were made, and the formalization "pushed back" one step from the surface structure to give a strick definition to what we now call the "message level" (McDonald & Meteer 1987). The new representational device that typifies this level is the "bundle specification" (section 2.2.2). Its original purpose was to impose functional and structural constraints on the attachment process; it has become a versatile facility for recording the intentions of a text planner.

As a virtual machine for linguistic realization, we consider Mumble-86 to be essentially completed as a research project. Certainly there are many extensions yet to be made to its grammar, and no doubt minor changes will be made from time to time to facilitate its use as a programming system. We do not expect, however, that these will necessitate any serious changes to the virtual machine proper. At this point, any factors that might influence us to make any serious changes would have to come from a deeper understanding of the nature of the internal representations that support the cognitive states that supply the information content that texts convey, and of the nature of intentions and goals and of how they are translated into actions within the generation process. This is where much of our new work is directed.

2. DESIGN

Contents:

2.1 OVERVIEW

MUMBLE is a linguistic realization component for natural language generation. It assumes that earlier components have determined the compositional structure of the utterance, the functional relations between the units, and choice of lexical heads. MUMBLE handles the constraints on realization (e.g. choosing between "the forces are attacking", "which are attacking", and "attacking" depending on whether a full clause, postmodifier, or premodifier is required), the assembly of all syntactic and morphological structures, and the maintenance of linguistic context.

2.1.1 Mumble's input

The input to Mumble is an explicit level of representation called *the message level* . It specifies what is to be said and constrains how it is to be said. The message level is made up of *realization specifications* written in Mumble's input specification language. Its minimal units are *kernel specifications*. Kernels are expressible as simple phrases in English (roughly equivalent to elementary trees in a Tree Adjoining Grammar). They have two main parts:

1) a realization function, which defines the possible surface forms of the phrase;

2) a list of arguments, which are themselves realization specifications;

Kernels compose into larger units, called bundle specifications. Bundles allow a planner to group information which can then be treated as a single unit and to express explicitly the relations among the units grouped. Bundles have three main parts:

1) the **head** is either a kernel or a bundle; it is realized first, as an "initial tree" into which other specifications are attached;

2) **further-specifications** have two parts, a specification (either a kernel or a bundle), which builds an "auxiliary tree", and an attachment function, which constrains where the new tree may be attached to the surface structure already built;

3) **accessories** contain information about syntactic details that are specific to natural language, such as tense, number, etc.

Different bundle types (e.g. GENERAL-CLAUSE, GENERAL-NP, DISCOURSE-UNIT, CONJUNCTION) have different drivers (the procedure used to process the bundle) and different accessories associated with them. For example GENERAL-CLAUSE has associated accessories TENSE-MODAL and QUESTION, whereas GENERAL-NP has accessories NUMBER and GENDER.

Figure 2.1 shows an example of the message level representation in the notation of our stand-alone interface for "Fluffy is chasing little mice in the basement"[1]:

[1] This example has been altered to make some points clearer: for a complete running example, see Figure 2.4 in Section 2.2 and Figure 7.1 in Section 7.

```
(general-clause
  :head (:realization-function CHASE/S-V-O-two-explicit-args
         :arguments (#<fluffy> #<little-mouses> ))
  :accessories (:tense-modal present  :progressive)
  :further-specifications
    ((:attachment-function clausal-adjunct
      :specifiation (location *self* #<basement> )
```

FIGURE 2.1

(Note: #<...> are abbreviations for objects in the underlying model of the applications program. In an actual input specification, they would be embedded message specifications. *self* stands for the entire bundle in which it appears.)

2.1.2 Mumble's internal representations and control structure

An underlying principle in Mumble's design is the use of multiple levels of explicit, executable representations and narrowly defined processes to map from one level to the other. There are two levels within Mumble in addition to the input specification and the output text: the surface structure and the word stream. These levels and the processes mapping between them are shown in figure 2.

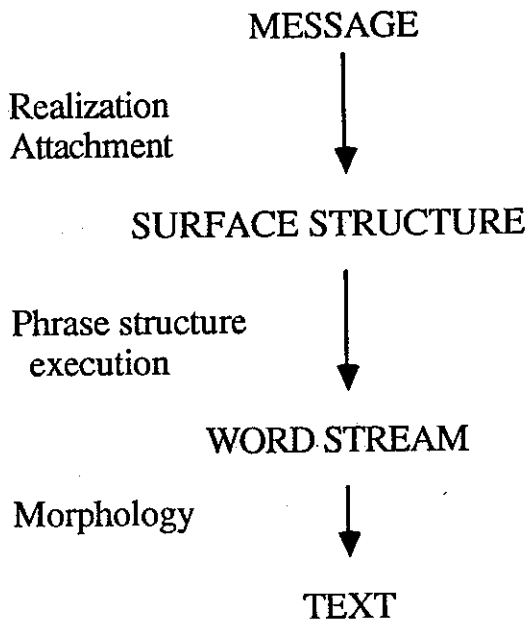


FIGURE 2.2

The processes *realization* and *attachment* transform the message level input to the intermediate representation, *the surface structure*. As a kernel specification is

realized, a particular phrase is chosen and built and the kernel's arguments are placed in the new structure. The choice of which phrase depends on the realization function of the kernel and the linguistic context. As a bundle is processed, the head is realized and the further specifications are spliced into the surface structure by the attachment process. Figure 3 shows the surface structure of the example after the bundle has been fully realized, but before any of its embedded arguments have been realized.

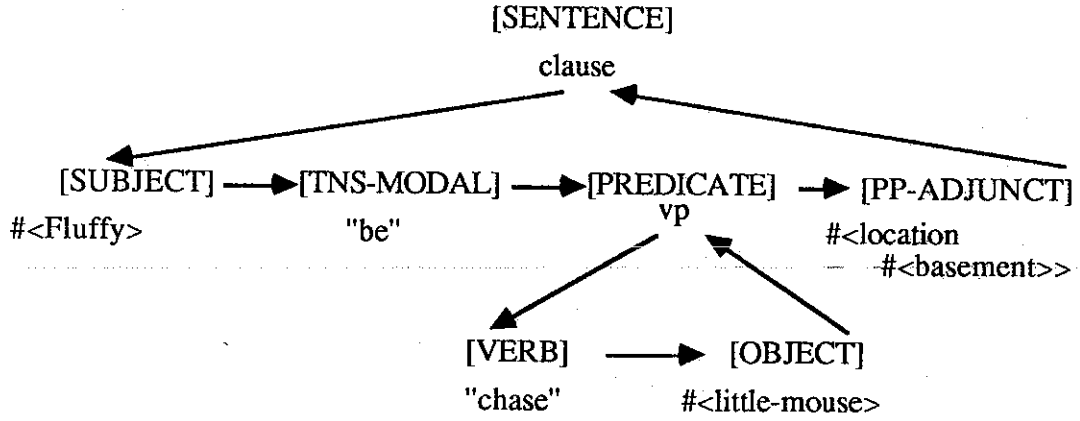


FIGURE 2.3

While surface structure is conventially represented as a tree, we use a more specialized representation as shown in Figure 2.3, which we call *position path notation.* It is distinguished from a tree in three ways:

1) The structure is a linked list of positions (the non-terminals) which amounts to a preorder traversal of a tree. Thus while sister nodes are linked, the tree structure itself (the direct association of daughter and parent nodes) is not represented.

2) Positions are annotated by one or more labels which carry grammatical constraints, specify actions to be carried out, and define where other phrases may be attached in with respect to that position.

3) Positions have "contents"; that is, a position directly dominates either a word, an unrealized specification (the leaves), or a rooted phrase.

Once the surface structure has been built by realization and attachment, it is then traversed depth first by the process *phrase structure execution*. This process guides the realization of embedded specifications and produces the word stream. The word stream representation goes through the morphological specialization process, which produces the final text.

2.2 MUMBLE'S INPUT SPECIFICATION LANGUAGE

Mumble's input is in the form of specifications in a particular input specification language. These *realization specifications* provide a set of instructions that direct the processes of realization and attachment. The minimal units are called *kernel specifications* and their composition into larger units are called *bundle specifications*. Figure 2.4 is an example of an input specification in the notation used by Mumble's "stand alone" interface (which allows the user to type explicit input specification for experimentation with Mumble).

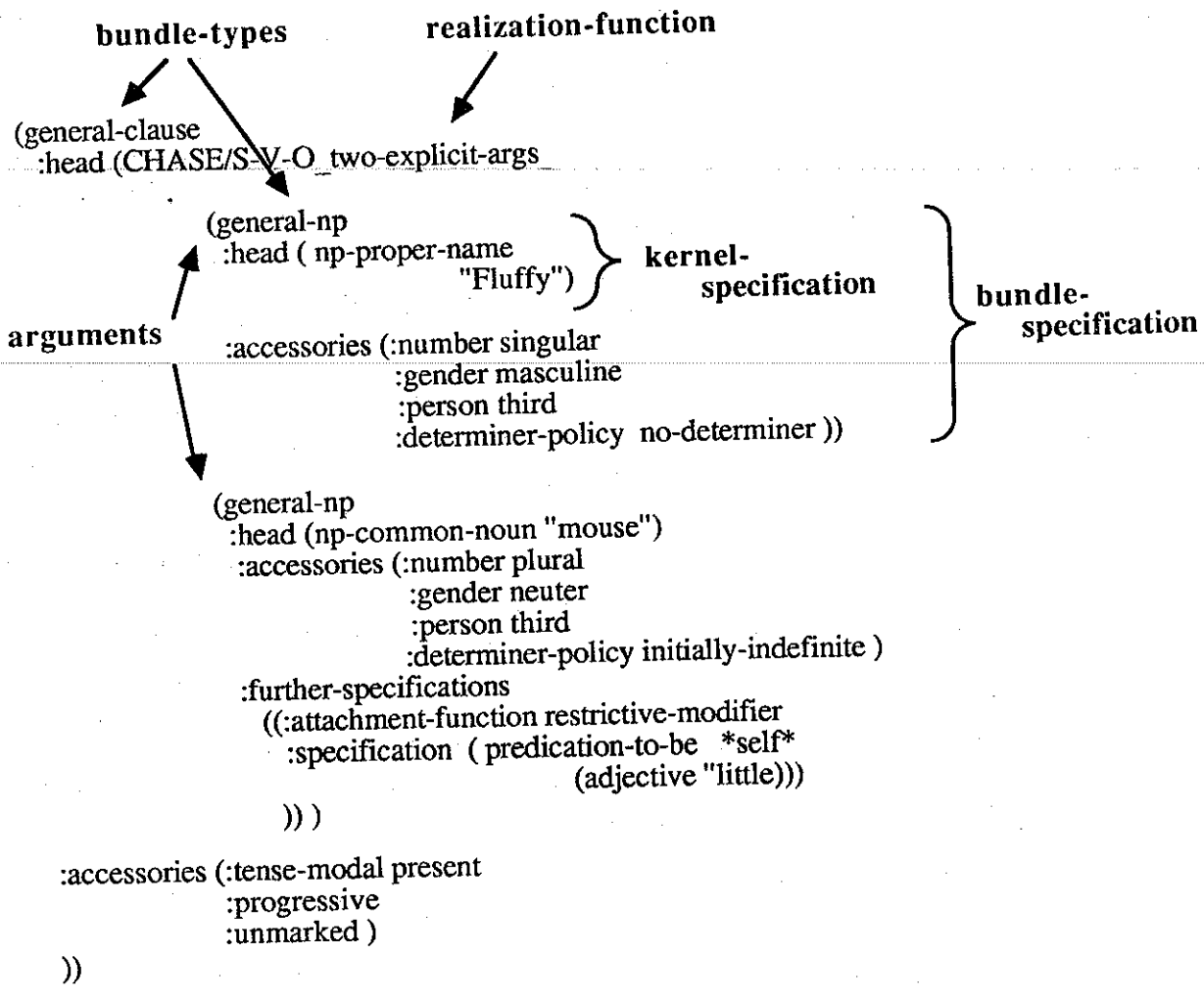


FIGURE 2.4

2.2.1 Kernel Specifications

Kernels represent the choice of a lexical head (e.g. "chase" or "mouse") and the specification of its arguments. This reflects our belief that one almost never chooses just to use a certain word in isolation, but rather to describe an action with a verb and simultaneous fixing of the arguments, or to describe an object by naming the natural kind it belongs to or by giving its name. (see also Kegl, 1987). The *arguments* of a kernel are to be applied to its realization function.

A kernel specification is to be interpreted (as are all devices at this level) as a function from what we would loosely call an "information unit" to a word or phrase at the surface linguistic level. By information unit we refer to the model level structured object or partial situation that the generator has chosen to realize by pairing it with this specification.

The realization function constrains the possible phrasal realizations of the head and arguments. Arguments are themselves realization specifications (bundles or kernels) or simply lexical items. (See examples in Figure 2.4.)

Realization functions

There are two types of realization functions: A REALIZATION CLASS specifies a set of possible phrases annoted by the characteristics that differentiate them. A SINGLE CHOICE specifies a particular phrase. (These will be discussed in detail in section 2.3) In general, the realization function is a class when there are multiple arguments which have alternative linearizations (order the arguments appear in the surface sentence); single choices are used when there is only a single argument (as in a noun phrase with only a head) or when there is only a single linearization (as in a prepositional phrase where the preposition always precedes the head). In the example in Figure 2.4 "chase" is the name of a realization class and "proper name" is the name of a single choice. As you can see, in the notation of the stand alone interface, they are indistinguishable; the difference is appreciated when the input specifications are processed by Mumble. The result when a realization function is run is always a simple phrase, comparable in size to an elementary tree in a Tree Adjoining Grammar. (We will elaborate more on this comparison and the details of TAGs which are relevant to our work in section 2.4).

2.2.1 BUNDLES AND THEIR ACCESSORIES

Bundles are compositional expressions which explicitly represent the functional relations between specifications and provide a structural context for specifying linguistically marked information such as tense, WH-extraction, or NP number. There are three major parts in a bundle:

1) the HEAD, which may be a kernel, bundle, or list of specifications; the head builds an "initial tree" (see discussion of TAGs xx) into which related specifications are attached;

2) a list of FURTHER SPECIFICATIONS, each of which is a kernel or a bundle specification marked with its role in the bundle; The specification results in an "auxiliary tree" (see discussion of TAGs xx) and the relation defines where it may be attached to the initial tree built by the head;

3) a list of ACCESSORIES, which carry the relevant syntactic details that are specific to natural language, such as tense, number, etc..

Bundles come in various types, each of which has its own driver and set of associated accessories. The driver is the procedure for processing the bundle parts. Accessories may be active or passive. Active accessories have associated processing procedures which run as part of the bundle's realization; for example, the accessory TENSE-MODAL splices a new slot in a tree and places its value (a tense or particular modal verb) in the slot's contents. Passive accessories are looked at when the bundle's head is realized as part of determining which phrase to choose. For example, if the accessory COMMAND is present, the command form (with the subject implicit, e.g. "Go home") will be chosen.

Bundle types. In this section, we describe each bundle type (particularly its driver and accessories) in detail. Since it is necessary to use a rather technical terminology that has not yet been defined, the reader may find it more helpful to skim this section at first, and then return to it for a closer perusal later.

general-clause

The driver for general-clause is very straight forward: it first realizes the head of the bundle, then processes the accessories (in a particular order so that interactions between, for example, tense-modal, which adds an auxiliary slot, and question, which inverts the auxiliary, are handled effectively) and then processes the further specifications in the order they appear. General clause has the following associated accessories:

TENSE-MODAL is an active accessory which takes a value: the tense marker *past* or *present*, or a modal verb (can, could, should, will, etc). When it runs, it splices in a new position after or before the subject (depending on whether the question accessory is present in the bundle) and puts the value in that position.

PERFECT and PROGRESSIVE are active accessories which splice in a new position marked appropriately as *have+en* or *be+ing* and place "have" or "be" in the

position respectively.[2] The actual morphological form of the verbs depends on the final combination and is handled later (see section 2.7).

QUESTION alters the position where the TENSE-MODAL slot will be spliced in.

NEGATION splices in a new slot directly after the tense carrier.

COMMAND, WH-QUESTION, and UNMARKED are passive accessories and are noticed by realization when choosing a phrase. WH-QUESTION takes a value, which is the argument to be questioned. They will be discussed in more detail in section 2.3.

We are experimenting with other accessories to mark arguments with such discourse features as GIVEN and EMPHASIZED, or to mark syntactically relevant arguments such as the central object in a purpose clause (see Huettner, Meteer (Vaughan), and McDonald, 1987).

general-np

The driver for general clause first checks to see if there is a reason the bundle should be realized as a pronoun rather than as a full np. (Reasons include C-command and obligatory reflexives: "Roscoe gave Floyd his fishing rod"; "Roscoe bought himself a new one".) If there is a reason, it uses the accessories on the bundle (NUMBER, GENDER, and PERSON) to choose a pronoun and place it as the contents of the current position. No noun phrase is built. If there isn't a reason to pronominalize, the actions taken are analogous to those above in general clause: first the head is realized, then the active accessories are processed, then the further specifications are attached in. Only NUMBER and DETERMINER-POLICY are processed actively and the processing of both involves setting a value in the state vector (see section x.x). Two examples of general-np bundles appear in the example shown in Figure 2.4.

conjunction

In a conjunction bundle, the head is a list of specifications, each of which will be a conjunct in the phrase. The bundle is agnostic as to the size or type of the elements (they may be kernels or bundles representing single words, noun phrases, or whole clauses). The driver builds a node and constituent slots, giving the slots the same grammatical constraints and other features as the slot dominating the whole conjoined phrase. It also puts comma labels on

[2]This analysis of the English auxiliary system follows the general outline of Chomsky's "aux-hopping" rule (1957).

appropriate slots and a conjunction label on the final slot. The particuar conjunction (and, but, or) is kept in the value of the CONJUNCTION accessory of the bundle. The example in Figure 2.5 shows an example of a conjunction bundle and the resulting phrase.

```
(conjunction-bundle
  :head ( (general-np
            :head  (proper-name "Floyd")
            :accessories (...) )
          (general-np
            :head  (proper-name "Helga")
            :accessories (...) ) )
  :accessories (:conjunction AND) )
```

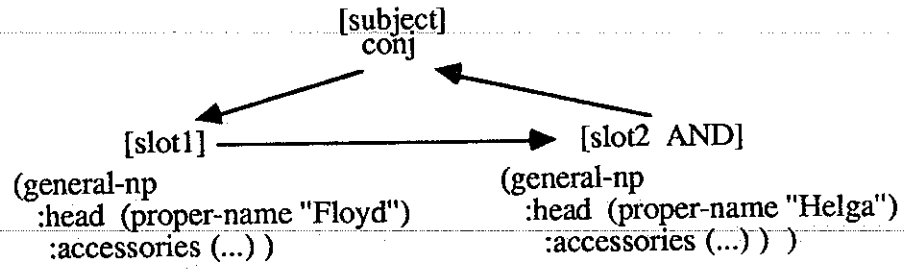


FIGURE 2.5

discourse unit

This bundle type is a means of grouping specifications together above the sentence level. It is currently very general, reflecting the need for more work on structure at this level. The driver builds a phrase with root DISCOURSE UNIT dominating a single slot SENTENCE. The head is placed as the contents of that slot and the further specifications are placed on a list of pending specifications to be realized later. The structure itself is agnostic as to whether these pending specifications are attached into the current sentence or attached as new sentences. While there currently are no accessories associated with this bundle type, a possible candidate would be initial interjections such as "well" or "now", which play a role in discourse structure but carry little or no content.

2.3 REALIZATION

The realization process transforms a realization specification into surface structure. As discussed earlier, a kernel specification is realized as a single phrase.

There are two parts to this process: choosing a phrase and instantiating that phrase. In section 2.3.1, we focus on the choice process. Realizing kernels and realizing bundle specifications are then addressed in section 2.3.2 and 2.3.3 respectively.

2.3.1 Realization Classes

The main choice mechanism in Mumble is the realization class, which is a predefined set of choices annotated by the characteristics that distinguish them. Choices in a class represent the possible surface realizations of a kernel specification in different linguistic contexts. The realization class for transitive verbs with two explicit arguments is shown in Figure 2.6. Each choice (numbered in this example)[3] consists of the name of a phrase and the parameters which map to the arguments of that phrase (see discussion of phrases in section 2.4) followed by its characteristics. For example the first choice is for SVO word order (AGENT VERB PATIENT are mapped to SUBJECT VERB OBJECT in the phrase) in the context of a main, unmarked clause. The second choice is for a subject relative clause. The order of the parameters for a relative clause puts AGENT in the COMP (wh) position and a trace of the AGENT in the SUBJECT position. Compare that with choice 3, which puts PATIENT in COMP and a trace in the OBJECT position.

[3] The numbers are only used here for reference; they are not part of the actual object.

```
(define-realization-class TRANSITIVE-VERB_TWO-EXPLICIT-ARGS (verb agent
                                                              patient)

      1  ((SVO  agent verb patient)
          :grammatical-characteristics (clause)
          :required-accessories (:unmarked))

      2  ((SVO-subj-rel  agent (agent :trace) verb patient)
          :grammatical-characteristics (relative-clause)
          :argument-characteristics (identical-with-root agent))

      3  ((Svo-obj-rel  patient  agent  verb  (patient :trace))
          :grammatical-characteristics (relative-clause)
          :argument-characteristics (identical-with-root patient))

      4  ((SVO-for-inf   agent verb patient)
          :grammatical-characteristics (for-infinitive))

      5  ((SVO-for-inf   (agent :trace) verb patient)
          :grammatical-characteristics (for-infinitive)
          :required-accessories (:purpose-clause-object agent))

      6  ((SVO-for-inf   (agent :trace) verb patient)
          :grammatical-characteristics (for-infinitive)
          :argument-characteristics (available agent))

      7  ((SVO-for-inf   agent verb (patient :trace))
          :grammatical-characteristics (for-infinitive)
          :required-accessories (:purpose-clause-object patient))

      8  ((SVO-for-inf   (agent :trace)  verb (patient :trace))
          :grammatical-characteristics (for-infinitive)
          :required-accessories (:purpose-clause-object patient)
          :argument-characteristics (available agent))

      9  ((SVO-subj-whq  agent (agent :trace) verb patient)
          :grammatical-characteristics (clause)
          :required-accessories (:wh agent))

     10  ((SVO-obj-whq  patient  agent  verb  (patient :trace))
          :grammatical-characteristics (clause)
          :required-accessories (:wh patient))

     11  ((SVO  (agent :trace) verb patient)
          :grammatical-characteristics (clause)
          :required-accessories (:command))
      )
```

FIGURE 2.6

There are three types of characteristics which annotate the choices:

Grammatical characteristics are compared with the grammatical constraints on the current position in the linguistic context. (The grammatical characteristics must be a subset of the constraints on the position for the choice

to be taken.) All choices have grammatical characteristics. Examples: NP, CLAUSE, RELATIVE CLAUSE.

Required accessories are compared with the accessories on the bundle being realized. Some, such as COMMAND, only look for the presence of the accessory in the bundle. Others, such as GIVEN, take an argument both with the characteristic and in the bundle. The accessory must be present in the bundle and its argument must be equal to the value of the parameter specified by the characteristic.

Argument characteristics have an associated function which takes as an argument the value of a parameter (which is an argument of the kernel being realized). For example IDENTICAL-WITH-ROOT(*agent*) checks to see if the value of *agent* is identical with the root of the phrase being attached to (e.g. when determining whether a relative clause should be [the book] "which Peter bought" or [Peter] "who bought the book".

Note that all choices do not necessarily have all types of characteristics. Almost all have grammatical characteristics (we will discuss an exception in a moment); choices that are dependent on some element already realized are more likely to have argument characteristics (such as a relative clause being attached to a noun phrase); choices dependent on information specified by the planner are more likely to have required accessories (such as a statement marked as a command or a particular argument marked as given or emphasized).

While choices are in general phrases (as in the above example), this is not the only type of allowable choice. As in choice 2 in the partial realization class shown in Figure 2.7, a choice might be simply one of the parameters in the class, or rather the *value* of that parameter when the class is called. This realization class would be used for constructions of the form "The dog is little", where the specification for "the dog" is mapped to **o** and the specification for "little" is mapped to parameter **p** ("be" is added at the level of the phrase). Choice 2 would be chosen when the specification is being attached to an np ("the dog") as a premodifier (checked by the argument characteristic), so the adjective alone would be contributed by this specification.

```
(define-realization-class PREDICATION_TO-BE (o p)

  1  ((S-be-COMP o p)
      :grammatical-characteristics (clause)
      :required-accessories (:unmarked))

  2  ((p)
      :argument-characteristics (identical-with-root o))

  3  ((S-be-COMP-subj-rel o (o :trace) p)
      :grammatical-characteristics (relative-clause)
      :argument-characteristics (identical-with-root o))
  . . . )
```

FIGURE 2.7

Note that in that case, there are no grammatical characteristics. Instead, the grammatical characteristics on the possible realizations of the value of the parameter are checked. This allows a more flexible system where the value of the parameter p need not be constrained to only be an adjective. For example, "The dog is little" and "The dog is in the parlor" can be handled by the same class.

2.3.2 Realizing Kernel Specifications

In this section we look at how realization classes and single choices are used in realizing kernel specifications. As discussed in section 2.2.1, a kernel specification consists of a realization function and a list of arguments. This realization function is in general a realization class (section 2.3.1) and the arguments of the kernel are mapped to the parameters of the class, becoming their values when the class is run. There are two other types of realization functions: curried realization classes and single choices.

Single choices. It is not always the case that there are multiple choices for the realization of a kernel. For example, a prepositional phrase is always realized the same way regardless of its linguistic context. In those cases, a SINGLE CHOICE is used instead of a realization class. A single choice (shown in Figure 2.8) consists of a phrase and the grammatical characteristics that indicate its correct use.

```
(define-single-choice NP-COMMON-NOUN
  :phrase  common-noun
  :grammatical-characteristics (np) )
```

FIGURE 2.8

The arguments of the kernel are mapped directly to the parameters of the phrase when the phrase is realized. The grammatical characteristics are included to prevent Mumble from producing an ungrammatical utterance if the single choice is incorrectly applied by the planner.

Curried realization classes are an experiment in exploring the nature of "words" in generation. Strictly speaking, they are not necessarily for MUMBLE-86's operation, though they are heavily used in our example "demos". Technically a curried realization class is a specification of a normal realization class by taking one of the parameters to the class (the verb in the cases currently implemented) and binding it to a particular word.

2.3.3 Realizing Bundle Specifications

Realization of bundles is driven by the particular bundle driver of that bundle type. The bundle driver determines the order of the realization of the head, realization of the further specifications, and the processessing of the accessories. (See discussion of particular types in section 2.2.1.)

The head of a bundle may be either a kernel or a bundle. If it is a kernel, it is realized in the manner described above; if it is a bundle, it is realized recursively using its bundle driver. However, the result of realization is always a single rooted phrase.

Only active accessories are actually processed while the bundle is being realized. Processing may involve splicing a new position into the tree or setting the state of the phrase (see section 2.2.1). The driver specifies the order in which the accessories should be processed with respect to each other and with respect to the realization of the head and further specifications. (For a detailed example, see Section 7.4.)

When further specifications are realized, the possible realizations of the specification begin attached are compared with the possible places where attachment can occur as defined by the attachment function. The intersection of these define the legal ways the specification may be attached. One attachment point is then chosen and new structure is spliced into the tree and the specification is "specialized" accordingly. (A specification is specialized by narrowing its realization function to include only the possibilities that are compatible with the particular attachment point chosen). For a more detailed discussion and example see section 2.6.

2.4 PHRASES AND POSITION PATH NOTATION

In this section we discuss the representation of the surface structure, which we represent in a particular notation called position path notation. We then discuss the building blocks of surface structure: phrases.

2.4.1 Position Path Notation

The surface structure that is built up by the processes of realization (section 2.3) and attachment (section 2.5) is represented in a particular notation we call *position path notation.* Rather than being a tree, as surface structure is conventionally represented, this structure is a linked list of positions. Positions may be one of two types:

NODES, which are the roots of phrases, and

SLOTS, which are the constituents of phrases.

Slots have contents, which may be an embedded phrase (in which case the root node of that phrase is the contents of the slot) or, at the leaves of the structure, may be a word or unrealized specification. It is important to remember that is this an evolving structure, with its growth governed by the process of Phrase Structure Execution (see Section 2.5), which traverses the surface structure. Figure 2.9 shows a snapshot of a surface structure with the links already traversed shown in bold.

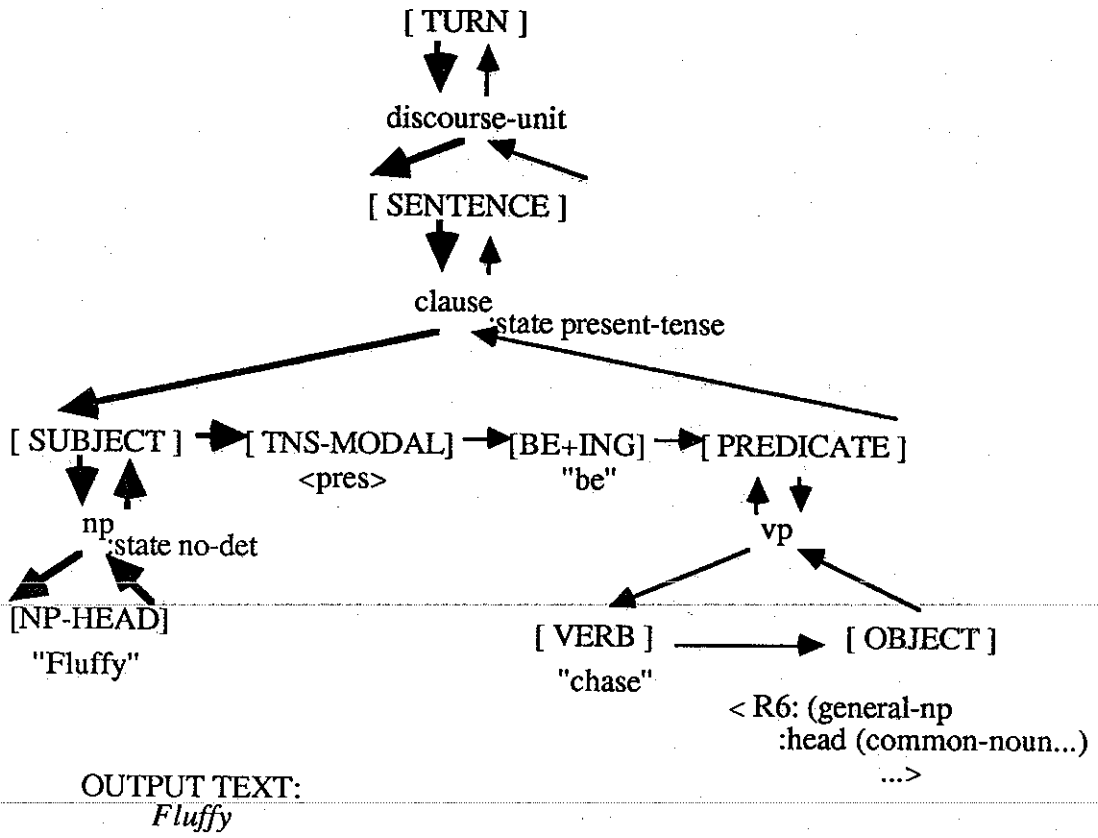


FIGURE 2.9

The major links between positions dictate the path of PSE.[4] Slots have a NEXT link (e.g. between SUBJECT and TENSE-MODAL) and a CONTENTS link (e.g. between SUBJECT and NP), which is defined only when the contents is a node. Nodes have a FIRST-CONSTITUENT link (e.g. between CLAUSE and SUBJECT) and a NEXT link (e.g. between CLAUSE and SENTENCE). We discuss the algorithm that follows the links (and the other actions PSE takes) in section 2.5.

The names on the positions indicate objects called LABELS. They carry information about the position and direct the activities at that position. (See examples of label definitions in Figure 2.10.) In particular, there are three types of information which may be associated with a label: grammatical-constraints, word-stream-actions, and associated-attachment-points. Each is discussed in detail below.

grammatical-constraints indicate the allowable categories of phrases that may be realized in that position, such as CLAUSE or NP (n.b. grammatical constraints are only found on slot labels). In the realization process they are

[4] The links shown represent the traversal path of PSE; there are also reverse links not shown which facilitate splicing new positions into the structure. See section 4.x for more detail.

compared with the GRAMMATICAL-CHARACTERISTICS on a choice in the realization process (see section 2.3.1)

word-stream-actions indicate direct actions to the output word stream. They are executed when PSE enters or leaves a position that the label annotates (depending on whether the word stream action is marked INITIAL or FINAL). The following are examples of each type of word-stream-action currently allowed:

(function-word initial "that") : prints the *pname* of the function word "that" when PSE enters the position for the first time;

(capitalize-next-word initial) : drops a *blip* in the word stream, which indicates the next word should be capitalized;

(punctuation final period) : prints the *pname* of the punctuation-mark PERIOD when PSE enters the position for the second time;

(determiner initial) : prints the appropriate determiner (depending on the STATE of the noun phrase, which carries information about the determiner policy that the planner specified).

associated-attachment-points is a list of the attachment points which are to be activated when instantiating a phrase which uses that label. Each attachment point has as part of its definition the link (NEXT or PREVIOUS) which is to be broken when splicing in a new position.

```
(define-slot-label SENTENCE
  grammatical-constraints (clause)
  word-stream-actions ((punctuation final period)
                       (capitalize-the-next-word initial)))

(define-slot-label SUBJECT
  grammatical-constraints (np)
  associated-attachment-points (tense-modal))

(define-node-label NP
  word-stream-actions ((determiner initial))
  associated-attachment-points (possessive))
```

FIGURE 2.10

2.4.2 Phrases

A phrase is a predefined specification of a portion of surface structure which, when instantiated, builds structure using position path notation (see 2.4.1). Phrases

correspond in size to the elementary trees in a Tree Adjoining Grammar. Complex surface structure is built up by attaching together simple phrases. (see section 2.6) Figure 2.11 shows the representation of a phrase in Mumble's grammar and the corresponding structure in path notation. A phrase consists of a list of parameters and a definition which identifies the root, constituent slots, the mapping of parameter values to positions, embedded nodes and constituents, and associated actions to be taken as the phrase is instantiated. Phrase definitions have the following syntax:

The first element in a list is always the label on the root of a phrase; (note embedded nodes such as VP appear as embedded lists in the definition).

The root may be followed by an optional set of keyword/value pairs which define actions to be taken when the phrase is instantiated (see below).

Next is a set of slot-label/contents pairs. The contents may be a parameter (in which case the value mapped to that parameter is placed as the slot's contents when the phrase is instantiated), a particular word (for example the word "be" in the auxiliary slot of a passive construction), or an embedded phrase (represented as a list). Each pair may also be followed by actions (see below).

Actions associated with a phrase: There are currently two actions that can be associated with phrase definitions:

:ADDITIONAL-LABELS (*list of labels*) adds those labels to the position (either a node or a slot; For example a subject position may get the addition label NOMINATIVE (*she* left) to govern case or may getFOR and OBJECTIVE if it is in a for-infinitive (*for her* to leave...) (See section 2.4.1 for a more complete discussion of the function of labels.)

:SET-STATE *value*, sets the state of the root of the phrase to that value, therefore it is only permissible after the root label. (For more information on the state of a root see section 2.7).

```
(define-phrase SVO   (S V O)
  (clause :set-state (:aux-state initial)
    subject S :additional-labels (nominative)
    predicate (vp
                verb V
                direct-object O  :additional-labels (objective))))
```

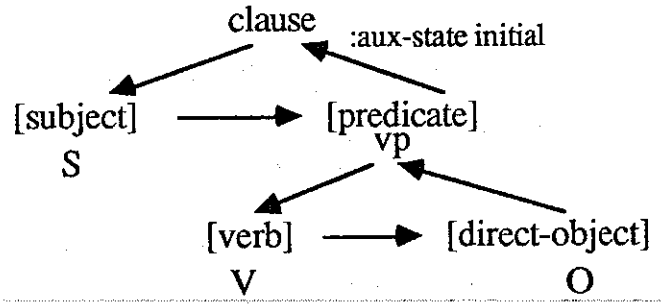


FIGURE 2.11

2.5 PHRASE STRUCTURE EXECUTION

Phrase structure execution (PSE) traverses the surface structure. Its actions are completely determined by the type of position (node or slot), the labels annotating it, and, in the case of slots, the type of its contents. Thus we can look at the surface structure representation as a high level language executed by the PSE process. Traversal is essentially preorder, so the order of the positions visited in the phrase shown in figure 2.11 would be as follows:

CLAUSE SUBJECT PREDICATE VP VERB DIRECT-OBJECT VP PREDICATE CLAUSE

2.5.1 The Traversal Algorithm

PSE uses the following algorithm to determine what actions to take and where to go next at any given point in the tree. There is a global variable CURRENT-POSITION which keeps track of the position of PSE.

If the current position is a slot, and this is the first time it has been visited by PSE then

1. execute any INITIAL word stream actions associated with the position. See discussion of word stream actions in section 2.4.1.)
2. a. if the contents of the slot is a word, send it off to morphology and say it (see section 2.7)
   b. if the contents is an unrealized specification, realize it (see section 2.3)

3. mark the position as visited
4. update current position: if the contents is a node, follow the contents link, otherwise follow the next link. (See discussion of links associated with position in section 2.4.1.)

If the current position is a slot, and it has already been visited by PSE then

1. execute any FINAL word stream actions associated with the position.
2. update current position: follow the NEXT link.

If the current position is a node, and this is the first time it has been visited by PSE then

1. execute any INITIAL word stream actions associated with the position.
2. mark the position as visited
3. update current position: follow the FIRST-CONSTITUENT link.

If the current position is a node, and it has already been visited by PSE then

1. execute any FINAL word stream actions associated with the position.
2. update current position: follow the next link.

2.6 ATTACHMENT

The attachment process allows the input to the generator to be compositional. As discussed above, minimal units in the input specification language, *kernels*, are expressible as simple phrases; larger utterances are formed by composing kernels: joining them together syntactically according to the relationships between them. These relationships will have been specifically selected as part of the text planning just as the realization function of a kernel is selected.

Realizing the relationship between two kernel units, like realizing the units themselves, is a matter of the generator selecting an appropriate linguistic device. The difference is that now the device is some combination pattern between elementary phrasal trees. In Tree Adjoining Grammar this operation is called *adjunction*; in Mumble-86 the operation differs sufficiently in its technical details that, given its independent origin, we refer to it as *attachment*. It is a linguistic operation that extends a surface structure tree at a specified point by the addition of another elementary tree.

In this section we first look at the ways specific attachment points are defined in Mumble; we then look at the grouping of attachment points into choice sets (attachment classes); finally, we look at how the attachment process makes use of these two structures.

2.6.1 Attachment points

An attachment point defines where more structure may be added to the tree. There are currently two types of attachment points: SPLICING ATTACHMENT POINTS define a link which is broken and a new slot added between the two positions that the link had connected. LOWERING ATTACHMENT POINTS define an entire phrase which is knit in between a slot and the node it dominates (effectively "lowering" some phrase already in the structure). We discuss each of these types in detail below.

Splicing attachment points are defined in terms of a label name (which identifies the position with respect to which structure will be attached), a link name (e.g. previous or next), and a label name to put on the new slot that will be spliced in. For example, in the attachment point shown in Figure 2.12, a slot labeled ADJECTIVE will be spliced into the PREVIOUS link of a position that has the label COMMON-NOUN.

```
(define-splicing-attachment-point ADJECTIVE
  reference-labels (common-noun)
  link (previous)
  new-slot (adjective))
```

FIGURE 2.12

Lowering attachment points specify a particular phrase that will be built when the attachment point is taken. One of its slots is designated as the KEY-POSITION. This is the position within the new phrase that where the phrase being attached to will be "lowered" into. Figure 2.13 shows a lowering attachment point which builds a new main clause

```
(define-lowering-attachment-point NEW-MAIN-CLAUSE
  new-phrase (svcomp)
  key-position (scomp))
```

FIGURE 2.13

This attachment point would be used in a case such as "Helga reported that little dogs chase mice". The verb "report" and its agent are added in as a further specification in the bundle and the action reported (the chasing) is the head of the bundle. The phrase built by the head is lowered into the complement position and a new main clause is attached in, as shown in Figure 2.14

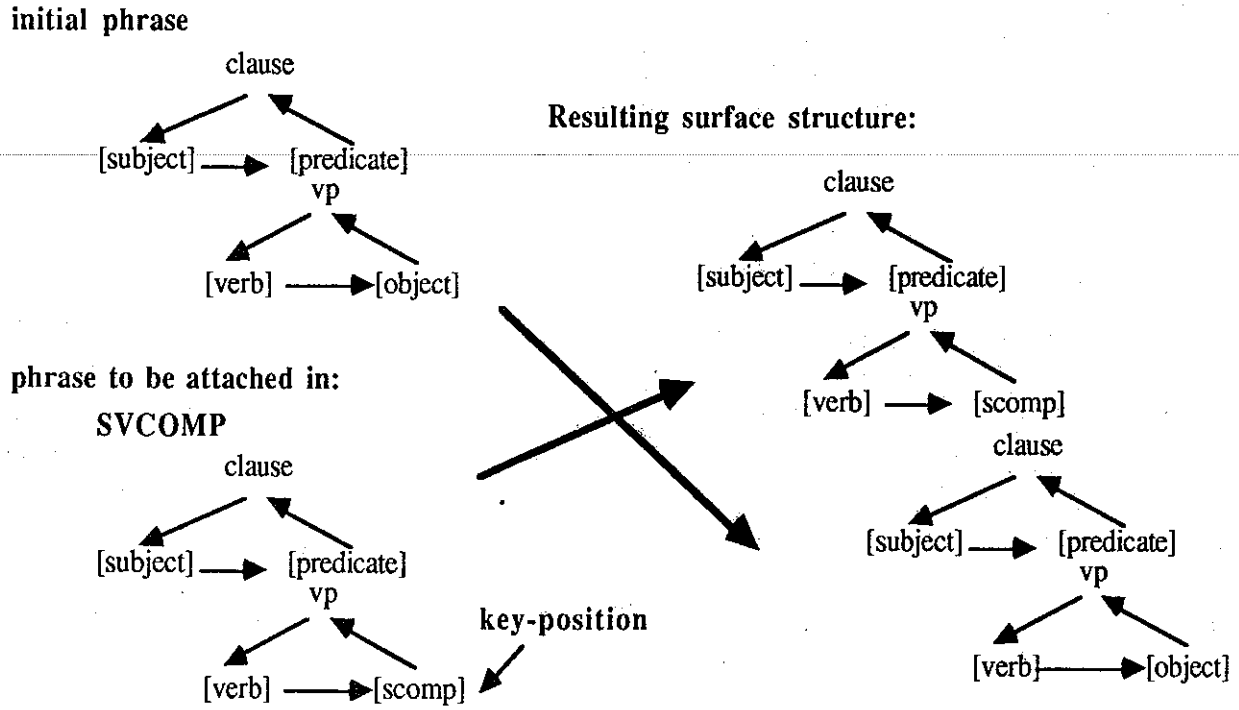


FIGURE 2.14

Attachment points may also have actions associated with them, which are run when the AP is taken. For example, the AP possessive has an associated action which changes the determiner state of the np, so if the determiner state is DEFINITE

(e.g. to produce "the dog") and a possessive slot is spliced in, the determiner state will go to NO-DETERMINER (e.g. to produce "Floyd's dog").

2.6.2 Attachment Classes

Attachment classes are an analogous control structure to realization classes, except that the choices are attachment points and they are distinguished on the basis of the possible realizations of the specification being attached in rather than by annotations on each choice.

Attachment points are grouped into classes based on a common functionality. For example, all the attachment points in the class shown in Figure 2.15 serves the function of restrictively modifying a noun. They differ in what link is broken and what syntactic category may be attached.

```
(define-attachment-class RESTRICTIVE-MODIFIER ( )
  ((ADJECTIVE)
   (RESTRICTIVE-APPOSITIVE)
   (RESTRICTIVE-RELATIVE-CLAUSE)
   (NP-PREP-COMPLEMENT )
  ))
```

FIGURE 2.15

2.6.3 The Attachment Process

In this section we look at one particular type of attachment: obligatory attachment within the processing of a bundle. This is the major type of attachment in Mumble, and the only type currently implemented.[1] As described in section 2.2.3, a bundle specification consists of a head, some accessories, and (optionally) further specifications. Each further specification consists of a specification and an attachment function which defines the possible places the specification may be attached. This attachment function may be either an attachment class, defining a set of points to be chosen between, or a particular attachment point (analogous to a single choice as a realization function).

The particular moment at which attachment occurs is dependent on the driver of the bundle type begin realized. In general that moment is after the head of the bundle has been realized, producing an initial tree. Next, all the attachment points associated with the labels on that phrase are gathered up into a table of attachment-point and position in the phrase. This is the ACTIVE-ATTACHMENT-POINT list. Now we are ready to attach the further specifications into the phrase. If the attachment

[1] Mumble is intended to include other types of attachment, such as optional attachment on the trailing edge of a phrase; however, since other types are not currently in the system, we will not describe them here.

function is an attachment class, a particular point must be first chosen from the set. The filtering process in in two steps. First, the APs in the class are compared with the active attachment point list, so that only those APs actually associated with the particular phrase being attached in to will be considered further. Next, the possible realizations of the specification being attached in are considered, so that only points that will allow the specification to be realized are considered. This filter uses the realization function of the specification, which will be either a realization class or a single choice. If it's a realization class, each choice in the class is checked: the grammatical characterists are compared with the grammatical constraints on the labels in the NEW-SLOT or KEY-POSITION of the attachment point (depending on whether it's a splicing or lowering AP) and the argument characteristics on each choice are checked to determine the relation of an argument with elements already realized (e.g. to determine whether a specification is modifying the head of the np it is being attached in to). (For a detailed example of how this works, see section 4.7.)

Once an attachment point is chosen, it must be attached into the surface structure. If it is a splicing attachment point, then the link identifed in the AP is broken and a new slot spliced in and the specification is placed as its contents.[2] If the attachment point is a lowering AP, the phrase indicated by the attachment point is instantiated and the arguments of the specification placed in it, then a subtree of the initial phrase becomes the contents of the slot marked as the key position, and the root of the new phrase is knit in where the subtree was. Figure 2.14 shows a special case of this where the root of the initial phrase is being lowered into the attached phrase using the attachment point NEW-MAIN-CLAUSE.

2.7 MORPHOLOGY AND THE WORD STREAM

The final level of representation in Mumble before the text is output to the screen is the word stream. Both the processes of PSE and Morphology contribute to transforming surface structure (with words in leaf position) to the word stream. PSE contributes directly through WORD-STREAM-ACTIONS (see Section 2.4.1) and indirectly by passing a word and its lingustic context (via the labels on its position) to the morphology process. This process performs the correct morphological specicalizations and sends the result to the word stream This level of representation is made explicit, rather than having morphology simply send the characters to the screen, so that interactions between adjacent words may be handled effectively. The

[2] To make the later realization of this specification more efficient, the realization function has been "specialized", that is narrowed to include only those choices found acceptable by the filtering process. This is done to avoid redundant effort, and has no effect on what the choice would be.

word stream keeps a two word buffer, so that, for example, "a" can be changed to "an" when followed by a word that begins with a vowel.

2.7.1 Morphology

The top level function of the morphology process, MORPHOLOGICALLY-SPECIALIZE-&-SAY-IT , is a dispatch off the labels on the current position and the labels on the word. (Note: one needs both to distinguish words that have more than one part of speech, such as "attack" which can be both a noun and a verb.) In this section we discuss the more complex morpholgoical routines for verbs and nouns.

Verb Group Morphology

The morphology of the verb group in Mumble is controlled by a finite state machine (shown in Figure 2.16). This type of control is particularly effective for English auxiliary and main verbs, since the particular morphological markings on any element of the verb group is dependent on the previous verb group element, and this information may be represented by a state. For example the progressive adds the auxiliary "be", which takes the tense of the verb group, and contributes the affix "-ing" to the verb group element after the "be". Notice, this element does not have to be next in the word stream ("He is finally leaving").

The actions taken by morphology and the transitions through the state machine are defined in terms of three elements:

1) the state of the current phrasal root (which for verb groups will generally be a clause;

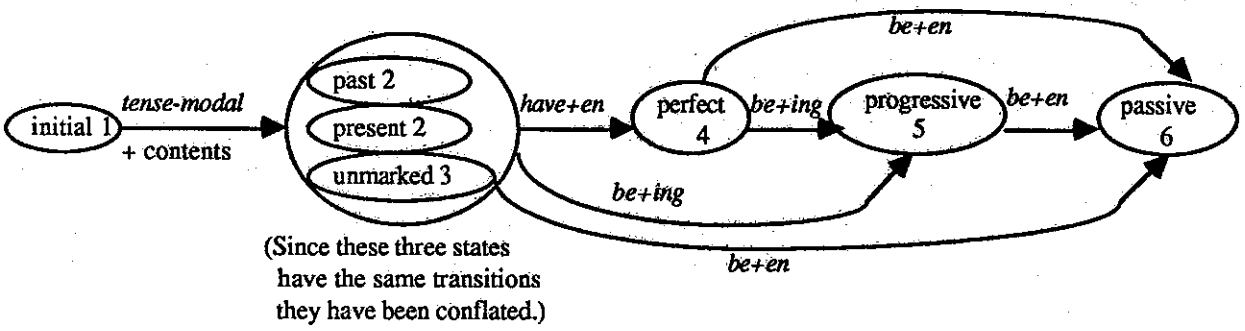
2) the label on the current position;

3) the contents of the current position (which will be a word or a tense marker).

The algorithm is as follows:

Execute the procedure associated with the state (see Figure 2.16); in general, this procedure has the effect of morphologically marking the word in a slot's contents and saying it.

Use the label on the current position to determine which state to go to and move to that state (i.e. change the value of the state of the phrasal root).



KEY:
Circles indicate states.
Numbers indicate procedures associated with the state.
Words in italics indicate the slot label on the current position.

NOTE: All states are final states.

FIGURE 2.16

Procedures associated with states (numbers correspond to states as indicated in Figure 2.16):

1. If contents is a modal verb, output it with no mophological specializations;
   if contents is a verb, output the infinitive marker "to" and then the verb with no morphological specializations;
   if the contents is a tense marker, do nothing
2. If the state is PRESENT, mark the contents with present tense morphology and output;
   if the state is PAST, mark the contents with past tense morphology and output;
3. Output the contents with no morphological specializations.
4. Output the en-form of the contents.
5. Output the ing-form of the contents.
6. Output the en-form of the contents.

Noun phrase morphology

Morphological specializations of nouns also uses information from the state, however, it is not as complex as that of the verb group since in English nouns are only marked with number. The state of the noun phrase is a vector with values for determiner policy (used by the word stream action on the node) and number. This information is kept at the phrasal root of the np, rather than for example with the word in the head position, so that agreement within the noun phrase can be handled. While this is rare in English, it does occur with demonstrative pronouns: "this book", "these books". Morphology accesses the value of number from the current phrasal root and either computes the plural form of the word or, in the case of irregular nouns, looks it up (see Figure 2.17).

```
(define-word "mouse" (noun) plural "mice")
```

FIGURE 2.17

Note that in other languages, especially those which mark nouns for case and gender, this process will be more complex. Gender information would also be kept in the state vector and case is obtainable from the slot dominating the noun phrase (e.g.the subject is marked with the label NOMINATIVE).

2.7.2 Pronouns

While much work has gone into theories of determining the anaphoric reference of a pronoun, little has gone into what criteria govern the use of pronouns. While we don't address the question directly in our system, we disctinguish two general categories of criteria: grammatical and discourse.

Grammatical criteria governs interogative and relative pronouns and required intrasentential pronominalization. The choice of when to use an interrogative or relative pronoun is governed by the the slot dominating that position, which will be explicitly marked with a label INTERROGATIVE-PRONOUN or RELATIVE-PRONOUN. Personal pronouns are required by the grammar when they are correferential with a noun phrase in the same sentence which "c-commands" them (to use standard linguistic terminology). In our system this is implemented with a "dominates and precedes" algorithm that is executed by the np-bundle driver (see section 2.x). An example of required pronominalization is the following pair of sentences:

1) Peter bought Peter a book.
2) Peter bought himself a book.

(1) is only grammatical if there are two Peters. If they are correferential, the second must be pronominalized, as in (2).

Discourse criteria governs optional, generally intersentential, pronominalization. Since this involves more global criteria and interacts with other lexical choices, we assume it is done by the planning component, rather than inside Mumble. For example, the following pair of texts are both grammatical; however, the use of a full noun phrase in (2) instead of a pronoun as in (1) allows more information (i.e. the pejorative view of Fluffy) to be communicated.

1) Fluffy broke into the garden again yesterday. He trampled all the flowers and chewed up my new pink flamingo.

2) Fluffy broke into the garden again yesterday. The lousy mut trampled all the flowers and chewed up my new pink flamingo.

Pronouns chosen by the planner appear in the input specification as objects of type PRONOUN, just as lexical items appear as objects of type WORD.

Once the choice of whether to pronominalize has been made, there is the choice of which pronoun to use. Personal pronouns reflect the person, number, and gender of the object they refer to (I, you, he, she). Interrogative pronouns reflect animacy (what, who). Many pronouns reflect the case of there position in the sentence (she, her, who, whom). (Note this is the only relic left of English's once extensive case inflections.)

The objects of type pronoun in Mumble reflect those features that are instrinsic to the object they represent (gender, number,etc). The choice of case is made when the pronoun is reached by phrase structure execution and passed to the morphology process and it is governed by the labels on the position. Figure 2.18 shows examples of a personal and relative/interrogative pronouns.

```
(define-pronoun third-person-singular-feminine
  person third
  number singular
  gender feminine
  cases (make-pronoun-cases
          :nominative "she"
          :objective "her"
          :genitive "hers"
          :refexive "herself"
          :possessive-np "hers" ))

(define-pronoun which
  person third
  number singular
  gender neuter
  cases "which" ))
```

FIGURE 2.18

3. IMPLEMENTATION

Contents

3.1 The Core Virtual Machine

The virtual machine consists of a set of interpreters that manipulate typed, structure objects. The interpreters and the definition of the object types in effect constitute the processor with its instruction set. One then writes a grammar for Mumble-86 by defining a set of objects (a set of subroutines as it were), then has the text planner construct individual programs (input specifications) to set the machine into action. Internally to the virtual machine, the execution of the input specification results in the construction of another program, the surface structure, whose interleaved execution ultimately results in the production of the output text stream.

In this section we describe the virtual machine as a computational object implemented in Lisp. In 3.1 we go through each of the object types in turn, giving the external syntax that the grammar writer would enter to define an object, and the internal form that is created from it when the definition is processed. Important relationships among related sets of objects are described as well. In 3.2 we describe the functions that make up the interpreters and trace through their operation.

ALL THE OBJECT TYPES INHERITANCE TREE

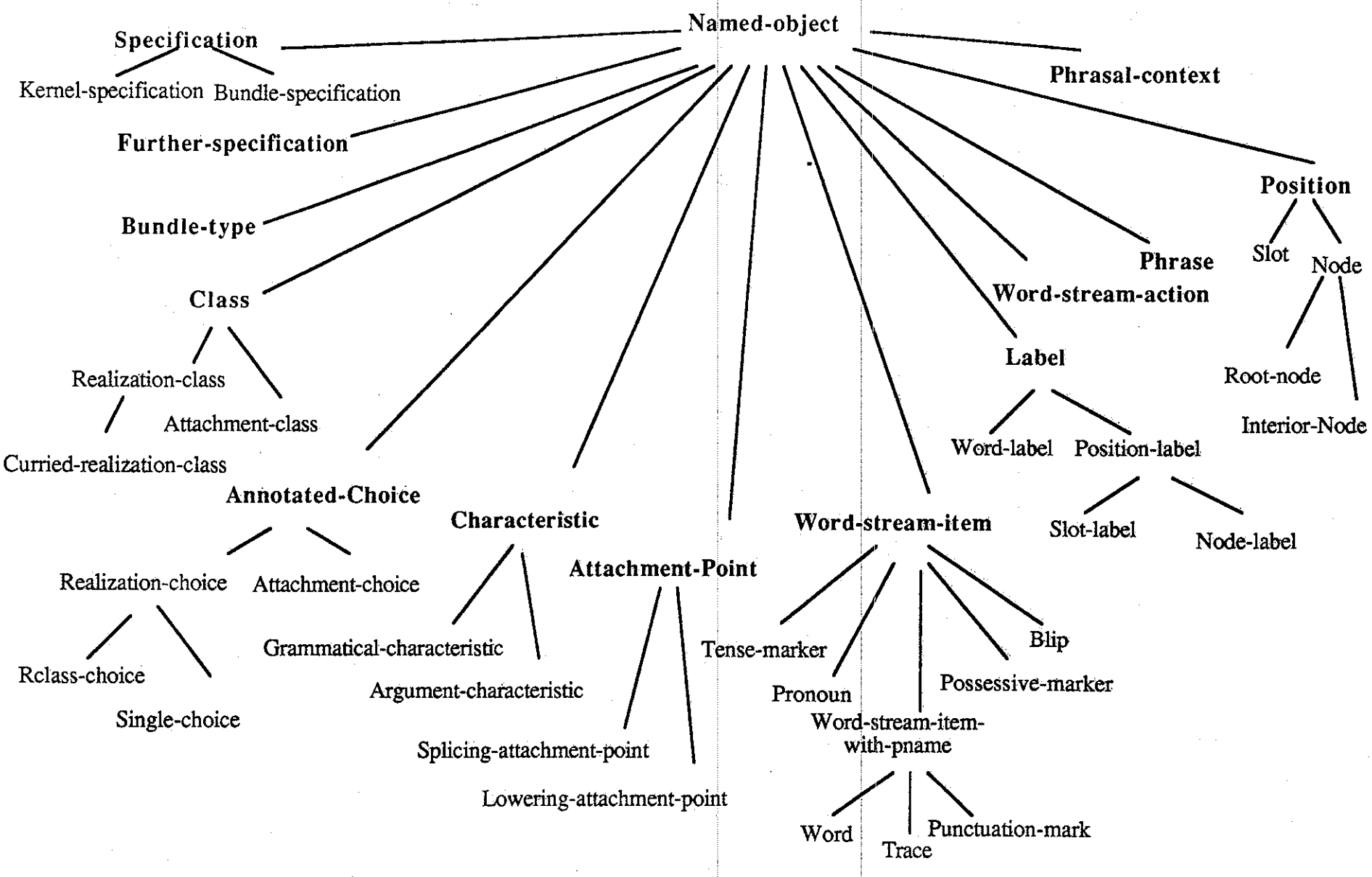


FIGURE 3.1

3.1.1 Object types

Mumble's object types are organized into a tree as shown in figure 3.1. This tree is in some respects an "ako" hierarchy: objects defined as types lower in the tree also belong to the types above them. Every object is a named-object; a root-node is also a node, a position, and a named-object. Semantically this hierarchy is defined by the addition of properties (specifically fields) as one proceeds down the tree. A root-node includes all of the fields of a node and adds one more. A node includes all of the fields of a position and adds two more; while a slot, the other daughter type of position, adds a different field.

The use of typed objects permits the interpreters to be implemented using simple and perspicuous Lisp forms like "typecase" to do its dispatches, and facilitates a number of technical matters as discussed in section 3.2.1. Organizing the objects into a hierarchy permits operations over whole groups of object types (e.g. "any kind of position") to also be done with type-based forms. It also has the technical benefit of allowing every field name to be uniquely associated with only one type.

Every object is given a name, usually automatically by copying or adapting a value from one of the object's other fields. The name is used in printing the object (using a convention analogous to the Commonlisp printed form for structures: "#< name-of-the-type name-of-the-object>"). Also the property list of the name symbol is where the type system (section 3.2.1) puts the information recording what objects share that name.

NAMED-OBJECT

Fields:	Value restrictions:
name	a symbol

Objects in Mumble may either be created at load time or at run time. Objects which make up the "reference knowledge" are predefined and created at postprocessing time when the system is loaded. Objects which are part of the representation of a particular utterance (i.e. the input specification and the surface structure) are built at run time. In what follows, we will go through the definitions of the object types grouped by functionality in the system, listing their fields and the restrictions on the field values in a table. For the objects that are predefined, we also include an example of an object of that type, illustrating the input syntax that one uses, and then show the internal form that that same object has after the input form has been processed. Note that many types are never instantiated directly (e.g. named-object), and consequently examples make no sense for them.

We begin with the objects that make up the input specifications (Section A). We then look at predefined classes and choices in the classes used in realization and attachment (Sections B-D). Next we look at phrases and labels (Section E) and the

objects that make up the surface structure representation (Section F). Finally, we look at the items in the word stream (Section G).

A. Realization Specifications

Realization specifications are the representation of the input to Mumble, and as such are created at execution time for the particular utterance. In our application interfaces (see section 6.1), they may be created by "templates" which are called from "default specifications". In our "stand alone" interface (see Section 6.2), they are built by the function CREATE-MESSAGE.

SPECIFICATION	includes: **named-object**
Fields:	value restrictions:
underlying-object	pointer to the object this specification is a mapping of

Kernel-specifications, the minimal units of the input specifications, and bundle-specifications, which are compositional structures, both inherit the UNDERLYING-OBJECT field of specifications, which keeps a point to the model level object which was the source of this specification. Note that further-specifications only appear as parts of bundles and only inherits from named object[1].

KERNEL-SPECIFICATION includes: **specification**

Fields:	value restrictions:
arguments	a list of specifications
realization-function	a realization-class, curried-realization-class, or a single-choice

```
#<KERNEL-SPECIFICATION CHASE> is a KERNEL-SPECIFICATION
   POSTPROCESSED?:  NIL
   UNDERLYING-OBJECT:  NIL
   ARGUMENTS:  (#<BUNDLE-SPECIFICATION GENERAL-NP for NP-PROPER-NAME>
                #<BUNDLE-SPECIFICATION GENERAL-NP for NP-COMMON-NOUN>)
   REALIZATION-FUNCTION:  #<CURRIED-REALIZATION-CLASS CHASE>
```

[1] The name field for specifications, further-specifications, and phrasal contexts is a new addition to the system and, thus are not reflected in the internal forms shown here which were produced before the change.

BUNDLE-SPECIFICATION includes: **specification**

Fields:	Value restrictions:
bundle-type	a bundle-type
head	a specification
accessories	a list of accessory-types
further-specifications	a list of further-specifications

```
#<BUNDLE-SPECIFICATION GENERAL-NP for NP-COMMON-NOUN> is a BUNDLE-
                                                      SPECIFICATION
      POSTPROCESSED?:  NIL
      UNDERLYING-OBJECT:  NIL
      BUNDLE-TYPE:  #<BUNDLE-TYPE GENERAL-NP>
      HEAD:  #<KERNEL-SPECIFICATION NP-COMMON-NOUN>
      ACCESSORIES:
   ((#<ACCESSORY-TYPE NUMBER> . #<ACCESSORY-VALUE SINGULAR>)
    (#<ACCESSORY-TYPE DETERMINER-POLICY> . #<ACCESSORY-VALUE KIND>))
      FURTHER-SPECIFICATIONS:  (#<FURTHER-SPECIFICATION RESTRICTIVE-
                                                      MODIFIER>)
```

FURTHER-SPECIFICATION includes: **named-object**

Fields:	value restrictions:
specification	a specification
attachment-function	an attachment point or an attachment class

```
#<FURTHER-SPECIFICATION RESTRICTIVE-MODIFIER> is a FURTHER-SPECIFICATION
  POSTPROCESSED?:  NIL
  SPECIFICATION:  #<BUNDLE-SPECIFICATION GENERAL-CLAUSE for ADJECTIVE>
  ATTACHMENT-FUNCTION:  #<ATTACHMENT-CLASS RESTRICTIVE-MODIFIER>
```

The following types, BUNDLE-TYPE, ACCESSORY-TYPE, and ACCESSORY-VALUE, are predefined objects used in specifications. For a discussion of the various bundle types and their associated accessories, see section 2.2.2.

BUNDLE-TYPE includes: **named-object**

Fields:	value restrictions:
driver	a symbol naming a function
accessory-list	a list of accessory-types

```
(define-bundle-type GENERAL-CLAUSE
                    clausal-bundle-driver
   accessory-list (:question
                   :tense-modal
                   :perfect
                   :progressive
                   :negate
             ;;the following are passive accessories (unordered)
                   :wh
                   :command
                   :purpose-clause-object
                   :given
                   :unmarked))

#<BUNDLE-TYPE GENERAL-CLAUSE> is a BUNDLE-TYPE
  POSTPROCESSED?:  T
  NAME:  GENERAL-CLAUSE
  DRIVER:  CLAUSAL-BUNDLE-DRIVER
  ACCESSORY-LIST: (#<ACCESSORY-TYPE QUESTION> #<ACCESSORY-TYPE TENSE-MODAL>
       #<ACCESSORY-TYPE PERFECT> #<ACCESSORY-TYPE PROGRESSIVE>
       #<ACCESSORY-TYPE NEGATE> #<ACCESSORY-TYPE WH>
       #<ACCESSORY-TYPE COMMAND> #<ACCESSORY-TYPE PURPOSE-CLAUSE-OBJECT>
       #<ACCESSORY-TYPE GIVEN> #<ACCESSORY-TYPE UNMARKED>)
```

ACCESSORY-TYPE includes: **named-object**

Fields:	value restrictions:
possible-values	a list of accessory values

```
(define-accessory-type :GENDER
   (masculine feminine neuter))

#<ACCESSORY-TYPE GENDER> is a ACCESSORY-TYPE
  POSTPROCESSED?:  T
  NAME:  :GENDER
  POSSIBLE-VALUES: (#<ACCESSORY-VALUE MASCULINE>
                    #<ACCESSORY-VALUE FEMININE>
                    #<ACCESSORY-VALUE NEUTER>)
```

ACCESSORY-VALUE includes: **named-object**

Fields:	value restrictions:
corresponding-type	an accessory-type

There is no define form for accessory-values. They are created automatically from the symbols that appear in the possible values field of the define form for accessory values.

```
#<ACCESSORY-VALUE MASCULINE> is a ACCESSORY-VALUE
   POSTPROCESSED?:  NIL
   NAME:  MASCULINE
   CORRESPONDING-TYPE:  #<ACCESSORY-TYPE GENDER>
```

B. Types used in classes

A class in Mumble is a predefined set of annotated choices. The types in this section exist to define the fields and subtypes that all classes share. The particular types of classes used for realization are shown in Section C and those for attachment in Section D.

CLASS	includes: **named-object**
Fields:	Value restrictions:
parameters	a list parameters
choices	a list of annotated-choices

ANNOTATED-CHOICE	includes: **named-object**
Fields:	Value restrictions:
choice	a phrase, attachment-point, parameter, etc.
argument-list	a list of parameters

CHARACTERISTIC includes: **named-object**

As with the type CLASS, the type CHARACTERISTIC will always be specialized before it is used.

PARAMETER includes: **named-object**

```
#<PARAMETER VERB> is a PARAMETER
   POSTPROCESSED?:  NIL
   NAME:  VERB
```

Parameters are the same type for all classes; they are not defined individually, but rather created as a side effect of creating classes.

C. Realization

The type REALIZATION-CLASS inherits all of its fields from CLASS. The choices are possible phrasal realizations for a specification. Note that the definition of a realization class includes the complete definition of each annotated choice with its characteristics. As you can see from the internal objects shown, the parameters and choices become separate first class objects when the class is postprocessed (see rclass-choice below).

REALIZATION-CLASS includes: class

```
(define-realization-class TRANSITIVE-VERB_TWO-EXPLICIT-ARGS (verb agent
                                                              patient)

        ((SVO  agent verb patient)
         :grammatical-characteristics (clause)
         :required-accessories (:unmarked))

        ((SVO-subj-rel  agent (agent :trace) verb patient)
         :grammatical-characteristics (relative-clause)
         :argument-characteristics (identical-with-root agent))

        ((Svo-obj-rel  patient  agent  verb  (patient :trace))
         :grammatical-characteristics (relative-clause)
         :argument-characteristics (identical-with-root patient))

        ((SVO-for-inf   agent verb patient)
         :grammatical-characteristics (for-infinitive))

        ((SVO-for-inf   (agent :trace) verb patient)
         :grammatical-characteristics (for-infinitive)
         :required-accessories (:purpose-clause-object agent))

        ((SVO-for-inf   (agent :trace) verb patient)
         :grammatical-characteristics (for-infinitive)
         :argument-characteristics (available agent))

        ((SVO-for-inf   agent verb (patient :trace))
         :grammatical-characteristics (for-infinitive)
         :required-accessories (:purpose-clause-object patient))

        ((SVO-for-inf   (agent :trace)  verb (patient :trace))
         :grammatical-characteristics (for-infinitive)
         :required-accessories (:purpose-clause-object patient)
         :argument-characteristics (available agent))

        ;;SVO-for-infinitives with gaps, such as for purpose clauses

        ((SVO-subj-whq  agent (agent :trace) verb patient)
         :grammatical-characteristics (clause)
         :required-accessories (:wh agent))

        ((SVO-obj-whq  patient  agent  verb  (patient :trace))
         :grammatical-characteristics (clause)
         :required-accessories (:wh patient))

        ((SVO  (agent :trace) verb patient)
         :grammatical-characteristics (clause)
         :required-accessories (:command))
        )
```

```
#<REALIZATION-CLASS TRANSITIVE-VERB_TWO-EXPLICIT-ARGS> is a REALIZATION-
                                        CLASS
   POSTPROCESSED?:  T
   NAME:  TRANSITIVE-VERB_TWO-EXPLICIT-ARGS
   PARAMETERS:  (#<PARAMETER VERB> #<PARAMETER AGENT> #<PARAMETER PATIENT>)
   CHOICES:  (#<RCLASS-CHOICE SVO> #<RCLASS-CHOICE SVO-SUBJ-REL>
              #<RCLASS-CHOICE SVO-OBJ-REL> #<RCLASS-CHOICE SVO-FOR-INF>
              #<RCLASS-CHOICE SVO-FOR-INF> #<RCLASS-CHOICE SVO-FOR-INF>
              #<RCLASS-CHOICE SVO-FOR-INF> #<RCLASS-CHOICE SVO-FOR-INF>
              #<RCLASS-CHOICE SVO-SUBJ-WHQ> #<RCLASS-CHOICE SVO-OBJ-WHQ>
              #<RCLASS-CHOICE SVO>)
```

CURRIED-REALIZATION-CLASS includes: **realization-class**

Fields:	value restrictions:
bindings	an alist of parameters and constant values
reference-realization-class	a realization-class

A Curried Realization class is simply a realization class with one or more parameters bound to values. Note the definition merely names a realization class, whereas the internal object includes all the choices from the class named.

```
(define-curried-realization-class CHASE (agent patient)
  Transitive-verb_two-explicit-args
  ((verb "chase")))

#<CURRIED-REALIZATION-CLASS CHASE> is a CURRIED-REALIZATION-CLASS
   POSTPROCESSED?:  T
   NAME:  CHASE
   PARAMETERS:  (#<PARAMETER AGENT> #<PARAMETER PATIENT>)
   CHOICES:  (#<RCLASS-CHOICE SVO> #<RCLASS-CHOICE SVO-SUBJ-REL>
              #<RCLASS-CHOICE SVO-OBJ-REL> #<RCLASS-CHOICE SVO-FOR-INF>
              #<RCLASS-CHOICE SVO-FOR-INF> #<RCLASS-CHOICE SVO-FOR-INF>
              #<RCLASS-CHOICE SVO-FOR-INF> #<RCLASS-CHOICE SVO-FOR-INF>
              #<RCLASS-CHOICE SVO-SUBJ-WHQ> #<RCLASS-CHOICE SVO-OBJ-WHQ>
              #<RCLASS-CHOICE SVO>)
   BINDINGS:  ((#<PARAMETER VERB> . #<WORD CHASE>))
   REFERENCE-REALIZATION-CLASS:
     #<REALIZATION-CLASS TRANSITIVE-VERB_TWO-EXPLICIT-ARGS>
```

REALIZATION-CHOICE includes: **annotated-choice**

Fields:	value restrictions:
grammatical-characteristics	a list of grammatical-characteristics
argument-characteristics	a list of argument-characteristics/argument pairs
required-accessories	a list of accessory and optional argument pairs

REALIZATION-CHOICE covers both RCLASS-CHOICE, which only occurs as part of a realization-class, and SINGLE-CHOICE, which can stand by itself. While these two types are postprocessed differently, they both inherit all of their fields from REALIZATION-CHOICE.

RCLASS-CHOICE includes: **realization-choice**

```
#<RCLASS-CHOICE SVO> is a RCLASS-CHOICE
  POSTPROCESSED?:  T
  NAME:  SVO
  CHOICE:  #<PHRASE SVO>
  ARGUMENT-LIST:  (#<PARAMETER AGENT> #<PARAMETER VERB> #<PARAMETER
                                                    PATIENT>)
  GRAMMATICAL-CHARACTERISTICS:  (#<GRAMMATICAL-CHARACTERISTIC CLAUSE>)
  ARGUMENT-CHARACTERISTICS:  NIL
  REQUIRED-ACCESSORIES:  (#<ACCESSORY-TYPE UNMARKED>)
```

SINGLE-CHOICE includes: **realization-choice**

```
(define-single-choice NP-PROPER-NAME
  :phrase  proper-name
  :grammatical-characteristics (np) )

#<SINGLE-CHOICE NP-PROPER-NAME> is a SINGLE-CHOICE
   POSTPROCESSED?:  T
   NAME:  NP-PROPER-NAME
   CHOICE:  #<PHRASE PROPER-NAME>
   ARGUMENT-LIST:  NIL
   GRAMMATICAL-CHARACTERISTICS:  (#<GRAMMATICAL-CHARACTERISTIC NP>)
   ARGUMENT-CHARACTERISTICS:  NIL
   REQUIRED-ACCESSORIES:  NIL
```

GRAMMATICAL-CHARACTERISTIC includes: **characteristic**

```
(define-grammatical-characteristic clause)

#<GRAMMATICAL-CHARACTERISTIC CLAUSE> is a GRAMMATICAL-CHARACTERISTIC
   POSTPROCESSED?:  NIL
   NAME:  CLAUSE
```

ARGUMENT-CHARACTERISTIC includes: **characteristic**

Fields:	value restrictions:
parameter	one of the parameters of the realization-class
consistency-test	a symbol naming a function

```
(define-argument-characteristic  IDENTICAL-WITH-ROOT (parameter)
  identical-with-root?)
```

```
   #<ARGUMENT-CHARACTERISTIC IDENTICAL-WITH-ROOT> is a ARGUMENT-
CHARACTERISTIC
      POSTPROCESSED?:  NIL
      NAME:  IDENTICAL-WITH-ROOT
      PARAMETER:  (PARAMETER)
      CONSISTENCY-TEST:  IDENTICAL-WITH-ROOT?
```

D. Attachment

Attachment classes are very similar to realization classes in that their definitions include definitions of the choices, which then become first class objects when postprocessed. The major difference is that in the current implementation, the choices do not have characteristics associated with them (see section 2.6 for a discussion of the process which choose between them). Note however, that there is nothing in the type system that disallows characteristics on attachment choices, we just haven't had an immediate need for them as yet.

ATTACHMENT-CLASS includes: **class**

```
(define-attachment-class RESTRICTIVE-MODIFIER (  )
  ((ADJECTIVE)
   (RESTRICTIVE-APPOSITIVE)
   (RESTRICTIVE-RELATIVE-CLAUSE)
   (NP-PREP-COMPLEMENT )))

#<ATTACHMENT-CLASS RESTRICTIVE-MODIFIER> is a ATTACHMENT-CLASS
   POSTPROCESSED?:  T
   NAME:  RESTRICTIVE-MODIFIER
   PARAMETERS:  NIL
   CHOICES:  (#<ATTACHMENT-CHOICE ADJECTIVE>
              #<ATTACHMENT-CHOICE RESTRICTIVE-APPOSITIVE>
              #<ATTACHMENT-CHOICE RESTRICTIVE-RELATIVE-CLAUSE>
              #<ATTACHMENT-CHOICE NP-PREP-COMPLEMENT>)
```

ATTACHMENT-CHOICE includes: **annotated-choice**

```
#<ATTACHMENT-CHOICE ADJECTIVE> is a ATTACHMENT-CHOICE
   POSTPROCESSED?:  NIL
   NAME:  ADJECTIVE
   CHOICE:  #<SPLICING-ATTACHMENT-POINT ADJECTIVE>
   ARGUMENT-LIST:  NIL
```

ATTACHMENT-POINT includes: **named-object**

Fields:	value restrictions:
actions	a list of actions to be carried out when this ap is used

There are two types of attachment point: splicing and lowering. We discuss the details of how these types differ and the attachment process in general in section 2.6.

SPLICING-ATTACHMENT-POINT includes: **attachment-point**

Fields:	value restrictions:
reference-labels	a list of labels
link	one of the symbols: previous, next, first, last-or last
new-slot	a label

```
(define-splicing-attachment-point ADJECTIVE
  reference-labels (nphead-cn)
  link (previous)
  new-slot (adjective))

#<SPLICING-ATTACHMENT-POINT ADJECTIVE> is a SPLICING-ATTACHMENT-POINT
  POSTPROCESSED?:  T
  NAME:  ADJECTIVE
  ACTIONS:  NIL
  REFERENCE-LABELS:  NPHEAD-CN
  LINK:  PREVIOUS
  NEW-SLOT:  #<SLOT-LABEL ADJECTIVE>
```

LOWERING-ATTACHMENT-POINT includes: **attachment-point**

Fields:	value restrictions:
new-phrase	a phrase
key-position	a label

```
(define-lowering-attachment-point NEW-MAIN-CLAUSE
  new-phrase (svscomp)
  key-position (scomp))

#<LOWERING-ATTACHMENT-POINT NEW-MAIN-CLAUSE> is a LOWERING-ATTACHMENT-POINT
  POSTPROCESSED?:  T
  NAME:  NEW-MAIN-CLAUSE
  ACTIONS:  NIL
  NEW-PHRASE:  #<PHRASE SVSCOMP>
  KEY-POSITION:  #<SLOT-LABEL SCOMP>
```

E. Surface Structure

Like specifications, the objects that comprise the surface structure are created while the system is running rather than having been predefined. We thus refer to them as "temporary" objects, and have provisions in the type system to eventually allow them to be deliberately thrown away after they have served their purpose. Surface structure objects are created as part of the action of the function "instantiate-phrase".

The only surface structure object types that are actually ever created are SLOTS, INTERIOR-NODES, and PHRASAL-ROOTS. The other types exist to define the fields that these types share, and to provide useful generalizations.

POSITION	includes: **named-object**
Fields:	value restrictions:
next	a position
previous	a position
labels	a list of labels
visited-status	one of the symbols: nil, entered, or left

By convention, the first label in the list of labels has special significance. Its name becomes the name of the position, and, more significantly, it is the only label that is presumed to supply any grammatical constraints on the realization of a slot's contents.

SLOT	includes: **position**
Fields:	value restrictions:
contents	a node, a specification, or a word

```
#<SLOT SENTENCE> is a SLOT
   POSTPROCESSED?:  NIL
   NAME:  SENTENCE
   NEXT:  #<PHRASAL-ROOT DISCOURSE-UNIT>
   PREVIOUS:  #<PHRASAL-ROOT DISCOURSE-UNIT>
   LABELS:  (#<SLOT-LABEL SENTENCE>)
   VISITED-STATUS:  LEFT
   CONTENTS:  #<PHRASAL-ROOT CLAUSE>
```

The presence of SLOTs in our model of surface structure is its most obvious point of distinction from the traditional representation. They have two principal roles (maintained by the diligence of the grammar writer): to describe the functional roles of constituents (e.g. "subject", or "theme"), and to uniquely name the position within the configuration of its root node.

NODE	includes: **position**
Fields:	value restrictions:
first-constituent	a slot
last-constituent	a slot

Nodes are distinguished according to whether they are the root node of a subtree (PHRASAL-ROOT) or one of its interior nodes (INTERIOR-NODE). Being a "subtree" of the surface structure is defined as being the result of instantiating one of the defined PHRASEs, which only comes about through the realization of a kernel specification. Consequently every subtree is the projection, onto the surface structure, of a specification at the message level.

INTERIOR-NODE includes: **node**

Notice that an INTERIOR-NODE adds no fields to the ones defined for NODE. This is quite deliberate, since it is the PHRASAL-ROOT that is the marked type of the pair. Adding this "trivial" type at the same level in the type hierarchy as PHRASAL-ROOT seemed to us to be a cleaner, less error-prone, design decision than to have just used the super-class NODE for this case.

```
#<INTERIOR-NODE VP> is a INTERIOR-NODE
   POSTPROCESSED?:  NIL
   NAME:  VP
   NEXT:  #<SLOT PREDICATE>
   PREVIOUS:  #<SLOT PREDICATE>
   LABELS:  (#<NODE-LABEL VP>)
   VISITED-STATUS:  LEFT
   FIRST-CONSTITUENT:  #<SLOT VERB>
   LAST-CONSTITUENT:  #<SLOT DIRECT-OBJECT>
```

PHRASAL-ROOT includes: **node**

Fields:	value restrictions:
context-object	a phrasal-context

```
#<PHRASAL-ROOT CLAUSE> is a PHRASAL-ROOT
  POSTPROCESSED?:  NIL
  NAME:  CLAUSE
  NEXT:  #<SLOT SENTENCE>
  PREVIOUS:  #<SLOT SENTENCE>
  LABELS:  (#<NODE-LABEL CLAUSE>)
  VISITED-STATUS:  LEFT
  FIRST-CONSTITUENT:  #<SLOT SUBJECT>
  LAST-CONSTITUENT:  #<SLOT PREDICATE>
  CONTEXT-OBJECT:  #<PHRASAL-CONTEXT for node CLAUSE>
```

Being the result of instantiating a PHRASE, the PHRASAL-ROOT object is a logical place to locate the tables of active attachment points, positions, and so on.

PHRASAL-CONTEXT

Fields:	value restrictions:
node	a node
position-table	a-list of position name and position
available-aps	a-list of attachment point and position
state	if clause node, then verb group state if np node, then noun phrase state
original-rspec	pointer to a specification this phrase was realized from

```
#<PHRASAL-CONTEXT for node CLAUSE> is a PHRASAL-CONTEXT
  POSTPROCESSED?:  NIL
  NODE:  #<PHRASAL-ROOT CLAUSE>
  POSITION-TABLE:  ((BE+ING . #<SLOT TENSE-MODAL>)
                    (TENSE-MARKER . #<SLOT TENSE-MODAL>)
                    (VP . #<NODE VP>)
                    (VERB . #<SLOT VERB>)
                    (DIRECT-OBJECT . #<SLOT DIRECT-OBJECT>)
                    (PREDICATE . #<SLOT PREDICATE>)
                    (SUBJECT . #<SL      OT SUBJECT>)
                    (CLAUSE . #<PHRASAL-ROOT CLAUSE>))
  AVAILABLE-APS:
   ((#<SPLICING-ATTACHMENT-POINT NEXT-AUX> . #<SLOT TENSE-MODAL>)
    (#<SPLICING-ATTACHMENT-POINT CLAUSE-PREP-COMPLEMENT>.
                                          #<PHRASAL-ROOT CLAUSE>)
    (#<SPLICING-ATTACHMENT-POINT WH-MARKER> . #<PHRASAL-ROOT CLAUSE>)
    (#<LOWERING-ATTACHMENT-POINT NEW-MAIN-CLAUSE> . #<PHRASAL-ROOT CLAUSE>)
    (#<SPLICING-ATTACHMENT-POINT TENSE-MODAL> . #<SLOT SUBJECT>)
    (#<SPLICING-ATTACHMENT-POINT PURPOSE-CLAUSE> . #<NODE VP>)
    (#<SPLICING-ATTACHMENT-POINT VP-PREP-COMPLEMENT> . #<NODE VP>))
  STATE:  ((:AUX-STATE . INITIAL))
  ORIGINAL-RSPEC:  #<BUNDLE-SPECIFICATION GENERAL-CLAUSE for CHASE>
```

F. Phrases and labels

Phrases are predefined objects used to build surface structure. They are roughly the size of elementary trees in a Tree Adjoining Grammar.

PHRASE includes: **named-object**

Fields:	value restrictions:
parameters-to-phrase	a list of paramters
definition	a list of labels, parameters, keywords, and lists

```
(define-phrase SVO  (S V O)
  (clause  :set-state (:aux-state initial)
     subject  S :additional-labels (nominative)
     predicate  (vp
                  verb V
                  direct-object O :additional-labels (objective))))
```

```
#<PHRASE SVO> is a PHRASE
  POSTPROCESSED?:  T
  NAME:  SVO
  PARAMETERS-TO-PHRASE: (#<PARAMETER S> #<PARAMETER V>#<PARAMETER O>)
  DEFINITION: ((#<NODE-LABEL CLAUSE> :SET-STATE (:AUX-STATE INITIAL)
  (#<SLOT-LABEL NOMINATIVE>) #<SLOT-LABEL PREDICATE>
  (#<NODE-LABEL VP> #<SLOT-LABEL VERB> #<PARAMETER V>
  #<SLOT-LABEL DIRECT-OBJECT> #<PARAMETER O>)))
```

LABEL includes: **named-object**

Labels are used for annotating other objects. Position labels annotate position in the surface structure. They are differentiated into node and slot positions, which both inherit the fields from POSITION-LABEL.

POSITION-LABEL includes: **label**

Fields:	value restrictions:
word-stream-actions	a list of word-stream-actions
associated-attachment-points	a list of attachment-points

NODE-LABEL includes: **position-label**

```
(define-node-label NP
  word-stream-actions ((determiner initial))
  associated-attachment-points (possessive))

#<NODE-LABEL NP> is a NODE-LABEL
  POSTPROCESSED?:  T
  NAME:  NP
  WORD-STREAM-ACTIONS:  (#<WORD-STREAM-ACTION DETERMINER>)
  ASSOCIATED-ATTACHMENT-POINTS:  (#<SPLICING-ATTACHMENT-POINT POSSESSIVE>)
```

SLOT-LABEL includes: **position-label**

Fields:	value restrictions:
grammatical-constraints	a list of grammatical-characteristics

```
(define-slot-label SENTENCE
   grammatical-constraints (clause)
   word-stream-actions ((punctuation final period)
                        (capitalize-the-next-word initial)))

#<SLOT-LABEL SENTENCE> is a SLOT-LABEL
  POSTPROCESSED?:  T
  NAME:  SENTENCE
  WORD-STREAM-ACTIONS:  (#<WORD-STREAM-ACTION PUNCTUATION>
                         #<WORD-STREAM-ACTION CAPITALIZE-THE-NEXT-WORD>)
  ASSOCIATED-ATTACHMENT-POINTS:  NIL
  GRAMMATICAL-CONSTRAINTS:  (#<GRAMMATICAL-CHARACTERISTIC CLAUSE>)
```

WORD-LABEL includes: **label**

Word labels essentially mark the syntactic category of words. They have no fields of their own.

```
(define-word-label NOUN)

#<WORD-LABEL NOUN> is a WORD-LABEL
  POSTPROCESSED?:  T
  NAME:  NOUN
```

WORD-STREAM-ACTION includes: **named-object**

Fields:	value restrictions:
moment	one of the symbols: initial or final
object	a word or punctuation-mark
condition	a symbol naming a function

Word stream actions are associated with position labels and are defined there, rather than being individually defined. They define some particular action taken at the level of the word stream when the position of the label is entered or left (depending on the value of the MOMENT field)

```
#<WORD-STREAM-ACTION CAPITALIZE-THE-NEXT-WORD> is a WORD-STREAM-ACTION
    POSTPROCESSED?:  T
    NAME:  CAPITALIZE-THE-NEXT-WORD
    MOMENT:  INITIAL
    OBJECT:  NIL
    CONDITION:  NIL
```

G. Word stream items

WORD-STREAM-ITEM includes: **named-object**

Word stream items are the objects that occur at the level of the word stream (see Section 2.7).

PRONOUN includes: **word-stream-item**

Fields:	value restrictions:
person	one of the symbols: first, second, or third
number	one of the symbols: singular or plural)
gender	nil, or one of the symbols: masculine, feminine, or neuter
cases	a structure, with a field for each possible English case. The value of the field is a quoted string to be sent to the output stream. For instance, `me' or `I.'

```
(define-pronoun third-person-singular-feminine
        person third
        number singular
        gender feminine
        cases  (make-pronoun-cases :nominative     "she"
                                   :objective      "her"
                                   :genitive       "hers"
                                   :reflexive      "herself"
                                   :possessive-np "hers"))
```

```
#<PRONOUN THIRD-PERSON-SINGULAR-FEMININE> is a PRONOUN
  POSTPROCESSED?:  NIL
  NAME:  THIRD-PERSON-SINGULAR-FEMININE
  PERSON:  THIRD
  NUMBER:  SINGULAR
  GENDER:  FEMININE
  CASES:  #S(PRONOUN-CASES :NOMINATIVE "she"
                           :OBJECTIVE "her"
                           :GENITIVE "hers"
                           :REFLEXIVE "herself"
                           :POSSESSIVE-NP "hers")
```

WORD-STREAM-ITEM-WITH-PNAME includes: **word-stream-item**

Fields:	value restrictions:
pname	a string

There are three types of word stream actions which are grouped under this subtype: words, punctuation marks and traces. They are similar in that they all have a PNAME field, which is what is printed in the word stream.

WORD includes: **word-stream-item-with-pname**

Fields:	value restrictions:
word-labels	a list of labels
rregularities	a detached plist

```
(define-word "mouse" (noun) plural "mice" )

#<WORD MOUSE> is a WORD
  POSTPROCESSED?:  T
  NAME:  MOUSE
  PNAME:  "mouse"
  WORD-LABELS:  (#<WORD-LABEL NOUN>)
  IRREGULARITIES:  (PLURAL "mice")
```

TRACE includes: **word-stream-item-with-pname**

Fields:	value restrictions:
original-specification	a realization-specification

Traces almost always have a vacuous pname, since they are most commonly used for gaps. The backpointer to the original specification allows us to do things like subject-verb agreement transparently. One trace with a pname is "there" (of "there is a book..."), which in our analysis is a trace. This allows "there" to remain in subject position and S-V agreement to be handled with a backpointer to the original specification as in other traces.

```
(define-trace wh-trace)
```

```
#<TRACE WH-TRACE> is a TRACE
  POSTPROCESSED?:  NIL
  NAME:  WH-TRACE
  PNAME:  NIL
  ORIGINAL-SPECIFICATION:  NIL
```

PUNCTUATION-MARK includes: **word-stream-item-with-pname**

```
(define-punctuation-mark PERIOD "." )

#<PUNCTUATION-MARK PERIOD> is a PUNCTUATION-MARK
  POSTPROCESSED?:  NIL
  NAME:  PERIOD
  PNAME:  "."
```

TENSE-MARKER includes: **word-stream-item**

```
(define-tense-marker PRESENT )

#<TENSE-MARKER PRESENT> is a TENSE-MARKER
  POSTPROCESSED?:  NIL
  NAME:  PRESENT
```

BLIP includes: **word-stream-item**

```
(define-blip  capitalize-the-next-word)

#<BLIP CAPITALIZE-THE-NEXT-WORD> is a BLIP
  POSTPROCESSED?:  NIL
  NAME:  CAPITALIZE-THE-NEXT-WORD
```

POSSESSIVE-MARKER includes: **word-stream-item**

```
(defvar the-possessive-marker
   (create-and-catalog 'the-possessive-marker
                       'possessive-marker
                       'name 'apostrophe-s)

#<POSSESSIVE-MARKER APOSTROPHE-S> is a POSSESSIVE-MARKER
  POSTPROCESSED?:  NIL
  NAME:  APOSTROPHE-S
```

3.1.2 INTERPRETERS

The interpreters of the virtual machine are divided into three interacting processes: Realization, which maps specifications into surface structures; Attachment, which extends to surface structure by attaching in more specifications; and Phrase Structure Execution, which traverses the expanding surface structure and thereby controls the machine as a whole.

3.1.2.a Phrase Structure Execution

As we've mentioned, descriptive linguists represents surface structure as trees, but Mumble uses a special "position path notation" (see Section 2.4). One principle difference between these representations is that our notation explicitly defines a left-to-right, depth-first path through the surface structure. It is this representation that Phrase Structure Execution traverses and therefore, it is not a recursive algorithm. In this section, we begin with the top level call to Mumble and trace the phrase structure execution algorithm.

The function MUMBLE is given a list of specifications.[1]

```
(defun MUMBLE  (list-of-rspecs)
  (let ((new-slot-for-this-turn
          (make-slot :name            'turn
                     :next            nil
                     :previous        nil
                     :labels          (list (slot-label-named 'turn))
                     :visited-status  'new
                     :contents        (car list-of-rspecs))))
    (setq pending-rspecs (cdr list-of-rspecs))
    (setq context-stack nil)
    (phrase-structure-execution new-slot-for-this-turn)))
```

The root of the whole surface structure is always a slot named "turn," and since we're starting off, we build that slot explicitly. (All other positions are built in the course of realization, as we will see below.) The first specification is made the contents of TURN and the rest are put aside for later (PENDING-RSPECS). The CONTEXT-STACK keeps a stack of the phrasal roots and is initialized to nil. We will see the usefulness of this global variable later. Finally, we begin Phrase Structure Execution at TURN.

The path that Phrase Structure Execution follows is made up of objects of type "position," that is, slots and nodes, and the implementation is just a simple loop over these objects until we get back to TURN, whereupon Mumble stops. Note the initializing of the global variable CURRENT-POSITION, which keeps a pointer to the

[1] All the functions presented here are excerpted from Mumble-86, but often have been simplified for clarity by deleting some lines of code.

location of the PSE. It will be idiosyncratically updated by the functions that process positions, PROCESS-NODE, and PROCESS-SLOT.

```
(defun PHRASE-STRUCTURE-EXECUTION (initial-position)
  (setq current-position initial-position)
  (until (PSE-finished? current-position initial-position)
    (etypecase current-position
      (node (process-node current-position))
      (slot (process-slot current-position))))

(defun PSE-FINISHED? (current-position initial-position)
  (and (eq current-position initial-position)
       (eq (visited-status initial-position) 'entered)))
```

Each position has a field called VISITED-STATUS, which is how Phrase Structure Execution knows whether it has been to a position before. The field starts out with the value 'NEW; this is changed to 'ENTERED when it is first encountered and 'LEFT when it is next encountered. No position will be encountered more than twice. We see the role of that status in the following processing functions, which are called by PHRASE-STRUCTURE-EXECUTION.

```
(defun PROCESS-NODE (position)
  (ecase  (visited-status position)
    (new
     (when (phrasal-rootp position)
       (entering-new-context position))
     (do-all-word-stream-actions (labels position) 'new)
     (set-visited-status position  'entered)
     (update-current-position (first-constituent  position )))

    (entered
     (when (phrasal-rootp position)
       (leaving-previous-context position))
     (do-all-word-stream-actions (labels position) 'entered)
     (set-visited-status position 'left)
     (update-current-position (next position)))))
```

```
(defun PROCESS-SLOT (position)
  (ecase  (visited-status position)
    (new
      (let ((contents (realization-cycle (contents position) position))
            (labels   (labels position)))
        (do-all-word-stream-actions labels 'new)
        (etypecase contents
          (word-stream-item
            (morphologically-specialize-&-say-it contents labels))
          (position
            (update-current-position contents))))
     (set-visited-status position 'entered))

    (entered
     (do-all-word-stream-actions (labels position) 'entered)
     (update-current-position (next position))
     (set-visited-status position 'left))))
```

These two functions, one for slots and one for nodes, each divides in two cases, one for entering the position and the other for leaving. Thus, there are four cases, but there is great similarity among them:

--They do all the word-stream-actions for the position. For instance, entering the position labeled SENTENCE will execute a word-stream-action to capitalize the next word. Similarly, leaving a position labeled SENTENCE will print a period. Other word-stream-actions put commas around appositives or print function words, such as the complementizer "that."

--They all update the visited-status of the position---NEW becomes ENTERED and ENTERED becomes LEFT.

--They update CURRENT-POSITION.

```
(defun UPDATE-CURRENT-POSITION (position)
 (setq current-position position))
```

Updating the current position is different in each of the four cases:

1) If we're entering a node, we begin a left-to-right traversal of its constituents, so the next position is the FIRST-CONSTITUENT field of the node.
2) If we're leaving a node, we follow the NEXT field, which we'll see is typically the slot it is the contents of.
3) If we're entering a slot, then if the contents is a word we say the word and keep the position the same (we'll immediately be leaving the position), and if the contents is a position (necessarily a node), we move down into that subtree, so the next position is the CONTENTS field of the slot.
4) If we're leaving a slot, we follow its NEXT field, which will either be another slot, (the next constituent in a sequence dominated by some node), or the node itself (if we are the last constituent).

Some nodes are phrasal roots, that is, they are the root of a phrase which was picked as the realization of some specification. Typically, these are phrases like NP or CLAUSE. As phrasal roots, they carry information about the phrase as a whole, such as a list of the positions or a list of the available attachment points. They also may carry state vectors, where, for instance, the number of a noun-phrase (singular or plural) or the auxiliary state of a verb-phrase are recorded (see Section 2.7). When Phrase Structure Execution enters a phrasal root, it makes this information globally accessible with the variable CURRENT-PHRASAL-ROOT, saving the previous phrasal root on the CONTEXT-STACK, which we saw initialized in the function MUMBLE. When Phrase Structure Execution leaves a phrasal root, it undoes these operations via following functions called from PROCESS-NODE:

```
(defun ENTERING-NEW-CONTEXT (position)
  (push current-phrasal-root context-stack)
  (setq current-phrasal-root (context-object position)))

(defun LEAVING-PREVIOUS-CONTEXT (position)
  (set-context-object position current-phrasal-root)
  (setf current-phrasal-root (pop context-stack)))
```

Finally, in the code for entering a slot, we see how Phrase Structure Execution controls Realization. Whenever we have enter a slot which has a specification as its contents, Realization is invoked to map that specification into structure. (Realization is discussed in section 3.1.2.b.

```
(defun REALIZATION-CYCLE (contents position)
  (if (specificationp contents)
      (let ((new-contents (realize contents)))
        (if (nodep new-contents)
            (knit-phrase-into-tree position new-contents)
            (set-contents position new-contents))
        (realization-cycle  new-contents position))
      contents))
```

Once Realization has mapped the specification into a phrase, we "knit" the phrase into the surface structure tree by setting the pointers necessary to make it the contents of the slot that held the specification. Sometimes, Attachment may have added additional surface structure to the left and right of the node that we are knitting into the tree. Consequently, the function below ensures that all new surface structure is knit it by searching to the left and right to get the FIRST and LAST position to be knit in.

```
(defun KNIT-PHRASE-INTO-TREE (containing-slot node)
  (let ((first (get-first-sibling-of-node node))
        (last  (get-last-sibling-of-node  node)))
    (set-previous first containing-slot)
    (set-next last containing-slot)
    (set-contents containing-slot first)
    ))
```

This knitting in of new phrases is why we said that Phrase Structure Execution traverses the *expanding* surface structure---Realization causes it to splice additional structure into its path.

3.1.2.b Realization

Realization maps specifications to phrases in a way which is sensitive to the linguistic context. We saw earlier that Phrase Structure execution maintains the linguistic context via the CURRENT-POSITION and the CURRENT-PHRASAL-ROOT, and that it calls the function REALIZE. Now we will see how REALIZE works and is constrained by that linguistic context.

```
(defun REALIZE  (realization-specification)
  (bind ((realization-specification realization-specification)
         (the-object-being-realized
           (underlying-object realization-specification)))
    (etypecase  realization-specification
      (bundle-specification
        (bind ((bundle-being-realized realization-specification))
          (funcall (driver (bundle-type realization-specification))
                   realization-specification)))
      (kernel-specification
        (realize-kernel-specification realization-specification)))))
```

After dynamically binding two variables which we will see used later, Realization splits in two for processing each of the two types of realization specifications: kernels and bundles. We will start our discussion with the realization of kernels and then discuss the realization of bundles.

Realizing Kernel Specifications

Each kernel specification has a "realization function". The first step is to distinguish between three kinds of realization functions: realization-classes, curried-realization-classes, and single-choices.

```
(defun REALIZE-KERNEL-SPECIFICATION (rspec)
  (let ((rfun (realization-function rspec)))
    (etypecase rfun
      (curried-realization-class
        (add-the-bindings-and-evaluate-its-realization-class rspec rfun))
      (realization-class
        (evaluate-its-realization-class rspec rfun))
      (single-choice
        (if (choice-is-acceptable rfun)
            (execute-choice (choice rfun) (arguments rspec) rspec)
            (mbug "Realization function ~s for ~s is not an acceptable choice"
                  rfun rspec))))))
```

All three kinds of realization functions eventually call the function EXECUTE-CHOICE. The first two have to first go through a process of figuring out which

choice to execute, while single-choices do not. Therefore, let's discuss the first two kinds of realization functions up to the point where they call EXECUTE-CHOICE and then cover all three by looking at EXECUTE-CHOICE.

A realization class has formal parameters, and these will be bound to the arguments of the kernel specification. A curried realization class is just like a realization class except that some of its arguments are bound *a priori*, rather than coming from the arguments to the kernel. So, we can easily discuss both simultaneously.

```
(defun ADD-THE-BINDINGS-AND-EVALUATE-ITS-REALIZATION-CLASS (rspec rclass)
  (let-with-dynamic-extent
        ((rclass-parameter-argument-list
               (create-rclass-parameter-argument-list-adding-curried-args
                 rclass
                 (parameters rclass)
                 (arguments rspec)
                 (bindings rclass) )))
    (select-and-evaluate-choice-from-rclass rspec rclass)))

(defun EVALUATE-ITS-REALIZATION-CLASS (rspec rclass)
  (let-with-dynamic-extent
        ((rclass-parameter-argument-list
               (create-rclass-parameter-argument-list
                 rclass
                 (parameters rclass)
                 (arguments rspec) )))
    (select-and-evaluate-choice-from-rclass rspec rclass)))
```

The first of these functions is for curried realization classes and the latter is for non-curried ones. They act nearly identically, binding the variable RCLASS-PARAMETER-ARGUMENT-LIST and calling the function SELECT-AND-EVALUATE-CHOICE-FROM-RCLASS. The variable is just a simple association list, a data structure which pairs each of the formal parameters of the realization class with one of the arguments of the kernel specification. The BINDINGS field of a curried realization class is also an association list, this time with of some formal parameters paired with the *a priori* values they are assigned. (Typically, this is verb. For instance, the curried realization class CHASE is the realization class TRANSITIVE-VERB-TWO-EXPLICIT-ARGS with the verb *a priori* bound to "chase.") Since the BINDINGS are already in association list format, they can be trivially added to the RCLASS-PARAMETER-ARGUMENT-LIST.

```
(defun SELECT-AND-EVALUATE-CHOICE-FROM-RCLASS (rspec rclass)
  (let* ((rclass   (realization-function rspec))
         (chosen   (choose-from-choice-set (choices rclass))))
    (execute-choice (choice chosen)
                    (argument-list chosen)
                    rspec)))
```

Having bound the arguments to the formal parameters, we can now procede to choose from the set of annotated choices in the realization class. Just a quick note about terminology: Each realization class has a set of choices, but each of these is combination of constraints on the choice (such as the grammatical characteristics and required accessories) and the choice itself (usually a phrase). We call each of these "choices." If we are trying to be very clear, the combination of a choice and its constraints is called an "annotated choice." This ambiguity of terminology is reflected in the code, because realization classes have a "choices" field, yet each of these objects has a "choice" field.

```
(defun CHOOSE-FROM-CHOICE-SET (set-of-choices)
  (cond (*ask-user-about-ambiguous-choice-sets?*
         (collect-all-useable-choices-and-ask-user set-of-choices))
        (*choice-selection-algorithm*
         (funcall *choice-selection-algorithm* set-of-choices))
        (T
         (take-the-first-acceptable-choice set-of-choices))))
```

For experimentation purposes, we may want to change the way that a particular annotated choice is selected, as you can see in the preceding function. We are interested here only in the last function.

```
(defun TAKE-THE-FIRST-ACCEPTABLE-CHOICE (alternative-choices)
  (or (some #'choice-is-acceptable alternative-choices)
      (mbug "None of the alternative choices were usable~%~A"
            alternative-choices)))

(defun CHOICE-IS-ACCEPTABLE (the-choice)
  (if (grammatical-characteristics-are-satisfied the-choice)
      (then
        (if (required-accessories-are-present (required-accessories the-
choice))
            (then the-choice)
            (else nil)))
      (else nil)))
```

As we can see, making the choice is relatively simple: we take the first annotated choice for which the grammatical characteristics are satisfied and the required accessories are present. Note that the function CHOICE-IS-ACCEPTABLE is called by REALIZE to ensure that a single-choice is not being used erroneously. Thus, the acceptability criteria for an annotated choice is identical for all three kinds of realization function. It is an error for no choice to be acceptable. Grammatical constraints are pretty simple. The variable CURRENT-POSITION that Phrase Structure Execution has managed for us will be bound to the slot that contains this specification. The first label on the slot is the one that gives the grammatical contraints, and we simply check that the grammatical characteristics of the annotated choice are a subset of the grammatical constraints on the position. The only complication comes when the choice is a parameter, which means the realization class would return one of its arguments (one of the arguments of the kernel specification).

This case can arise in the realization class PREDICATION_TO-BE, where, for instance, if the argument is a single-choice for the adjective "little," we might simply return that single-choice.

Of course, we have to ensure that the choice is reasonable in this linguistic context. That is done essentially recursively, so we will not explore it further.

```
(defun GRAMMATICAL-CHARACTERISTICS-ARE-SATISFIED (the-choice)
  (let ((grammatical-label (car (labels current-position))))
    (let ((constraining-characteristics
            (grammatical-constraints grammatical-label))
          (characteristics-of-the-choice
            (grammatical-characteristics the-choice)))
      (if (parameterp (choice the-choice))
          (check-if-rclass-of-parameter-is-compatible
            (choice the-choice)
            grammatical-label)
          (subsetp characteristics-of-the-choice
                   constraining-characteristics)))))
```

The other thing that makes a choice acceptable is having the correct required accessories. For this, we simply iterate over the required-accessories (a field of the annotated choice), and check whether each is mentioned in the BUNDLE-BEING-REALIZED, one of the variables dynamically bound by the function REALIZE. (Note that this reference to a bundle is made during the realization of a kernel, which means that any kernel whose realization function refers to required accessories must be contained in a bundle.) This check of required-accessories is complicated by the need to match the values of accessories that take values, such as WH. Consequently, that code has been omitted from this discussion.

At this point, Realization has looked at each of the annotated choices until it has found one which passes the tests of the grammatical constraints and required accessories. It will then extract the CHOICE field of the annotated choice and call the function EXECUTE-CHOICE, so the following discussion also applies to the realization of single-choices.

```
(defun EXECUTE-CHOICE (C arguments rspec)
  (etypecase C
    ((or word pronoun)  C)
    (parameter (return-rclass-parameter-value C))
    (symbol (if (eq c :choice-is-argument)
                (car arguments)
                (mbug "~A not a valid symbol or keyword" C)))
    (phrase (let ((result (instantiate-phrase C arguments)))
              (set-original-rspec (context-object result) rspec)
              result))
    (realization-class
      (mbreak "first example of a realization class as a choice"))))
```

A choice can be any of several kinds of objects, each of them listed here. If a choice is a word or pronoun, it is simply returned, and will be deposited in the

current-position. If it is a formal parameter, the value of that parameter is looked up (via the association list RCLASS-PARAMETER-ARGUMENT-LIST) and returned. For single-choices only, the choice can be the symbol :CHOICE-IS-ARGUMENT, in which case the argument is returned (the first in a list which will have only one element).

Most times, the choice will be a phrase, in which case it must be instantiated. Phrases are internally stored as a carefully structured list of labels and parameters. The arguments of the annotated choice are mapped one-to-one onto the parameters of the phrase. The S-expression is instantiated via the following rules:

--the CAR of a list is a node-label. A node will be built and the node-label put on it.

--the CDR of a list are the constituents of that node: even-numbered elements are slots and the element following each is its contents. The contents can be a list, in which case an embedded node structure is recursively built. If the contents is one of the parameters of the phrase, it is looked up via the PHRASE-PARAMETER-ARGUMENT-LIST (an association list entirely analogous tothe RCLASS-PARAMETER-ARGUMENT-LIST) and the argument placed in the slot. Thus, the arguments of the kernel specification are mapped onto the formals o the realization class, which are sent as arguments of the phrase (to be mapped onto the phrase's formal parameters), and finally deposited in slots of the phrase.

--any position can be followed by the keyword :additional-labels and a list of labels which are to be added to that position. This information is ignored as far as the first two rules are concerned.

--the top-most node is a phrasal root, and can be followed by information to initialize its state-vector. This is notated with the keyword :set-state and is followed by a plist of states and values, such as (:AUX-STATE INITIAL). Again, this information is ignored as far as the first two rules are concerned.

As the instantiating of a phrase is a straightforward recursive implementation of these rules, the code will be omitted here (see the function INSTANTIATE-PHRASE). The instantiated phrase is returned by the function REALIZE and knitted into the surface structure by Phrase Structure Execution.

Realizing Bundle Specifications

The realization of bundles depends entirely upon the particular bundle driver, though there are uniformities in how they run. The basic scheme is as follows: 1) realize the head (this is just a recursive call to the REALIZE function), which is typically a kernel; 2) process the accessories, which may splice in additional structure or affect the state-vector of the root (the phrase returned by realizing the head); and, finally, attaching in each of the further-specifications. Therefore, the realization of bundles invokes Attachment.

3.1.2.c Attachment

The bundle-driver that is realizing a bundle will process the further specifications in that bundle (an exception is when the bundle-driver for GENERAL-NP decides to pronominalize), calling the following function.

```
(defun PROCESS-FURTHER-SPECIFICATIONS (the-further-specifications)
   (dolist (fspec the-further-specifications)
      (attach (specification fspec)
              (attachment-function fspec))))
```

That is how the Attachment process is invoked. Attachment is given a specification---an ordinary kernel or bundle---and an "attachment-function." An attachment-function can either be a pre-selected attachment point or an attachment class, which is just a choice set of attachment points. If the latter, we go through a selection process and use the first acceptable one, while if the former, we need only validate the choice. Each of these finally calls ATTACH-AT-ATTACHMENT-POINT which does the modifications to the surface structure. We will discuss the processing of each kind of attachment-functionand then discuss its implementation of that function.

```
(defun ATTACH (rspec attachment-function)
  (let ((usable-choices
         (etypecase attachment-function
           (attachment-point
              (check-validity-and-attach-at-attachment-point
                attachment-function rspec))
           (attachment-class
              (choose-ap-and-attach attachment-function rspec)))))

    (specialize-realization-function rspec (nreverse usable-choices))
    ))
```

The following function checks the validity of a single attachment point.

```
(defun CHECK-VALIDITY-AND-ATTACH-AT-ATTACHMENT-POINT (ap rspec)
  (let ((active-ap (assoc ap (available-aps current-phrasal-root))))
    (unless active-ap
      (mbug "attachment point ~a not active" ap))

    (let ((usable-choices-for-rspec
           (choices-compatible-with-attachment-point ap rspec)))
      (unless usable-choices-for-rspec
        (mbug "None of the choices in the realization-function for ~a~%~
               are compatible with attaching it at ~a"
              rspec ap))

      (attach-at-attachment-point ap rspec (cdr active-ap))
      usable-choices-for-rspec)))
```

The global variable CURRENT-PHRASAL-ROOT is part of the linguistic context maintained by Phrase Structure Execution. It is bound to an object of type "phrasal-context," which contains tables of information about the phrase that we are traversing. One of those tables is of "available attachment points," and is implemented as an association list pairing attachment points with positions. (Since it points to real positions, this list is created when the phrase is instantiated. See the function INSTANTIATE-PHRASE.) This association list is stored in the AVAILABLE-APS field of the context-object which is the value of CURRENT-PHRASAL-ROOT.

If the attachment point we want to use is available, then we have a second check to make. Using the attachment point would result in a new slot being added to the surface structure with its contents typically the specification we are attaching in. We need to check that the specification can, in fact, be realized in that slot. We do this second check by running the filtering test used in Realization and seeing if any choices pass them. If none do, it is an error; otherwise we call ATTACH-AT-ATTACHMENT-POINT, and return the choices that were acceptable so that we can eliminate redundant work later (see call to SPECIALIZE-REALIZATION-FUNCTION within ATTACH).

```
(defun CHOICES-COMPATIBLE-WITH-ATTACHMENT-POINT  (ap rspec)
  (flet ((dispatch-and-check (ap candidate-choices)
           (etypecase ap
             (splicing-attachment-point
              (filter-on-grammatical-and-argument-characteristics
                candidate-choices (new-slot ap)))
             (lowering-attachment-point
              (filter-on-matching-phrase
                candidate-choices (new-phrase ap)))
             )))

    (multiple-value-bind (candidate-choices environment rclass rclass-context)
        (choices-relevant-to-attachment rspec)
      (ecase  environment
        (rclass (bind ((rclass-parameter-argument-list
                        (create-rclass-parameter-argument-list
                          environment
                          (parameters rclass)
                          (arguments rclass-context))))
                  (dispatch-and-check ap candidate-choices)))
        (single-choice
          (dispatch-and-check ap candidate-choices)) ))))

(defun CHOICES-RELEVANT-TO-ATTACHMENT (rspec)
  (etypecase rspec
    (bundle-specification
       (choices-relevant-to-attachment (head rspec)))
    (kernel-specification
       (let ((rfn (realization-function rspec)))
         (etypecase rfn
          (realization-class (values (choices rfn) 'rclass        rfn rspec))
          (single-choice     (values (list rfn)    'single-choice nil rspec))
))
 )))
```

The first step in this filtering is to track down the kernel specification that would be realized in this slot. Bundles don't matter because, while they may add structure into the subtree that is built for the head or modify its state-vector, they will not affect the labels on its root. Thus, in the code above, we start with a call to CHOICES-RELEVANT-TO-ATTACHMENT, which recursively searches down the specification to find a kernel. It then returns the relevant information about this kernel, which is the set of possible choices and information about the kernel. Back in the function CHOICES-COMPATIBLE-WITH-ATTACHMENT-POINT, we call the embedded subroutine DISPATCH-AND-CHECK to filter this kernel.

DISPATCH-AND-CHECK has to take the type of the attachment point into account in considering the possible realizations. If it is a "splicing attachment point," then the choices are filtered by grammatical and argument characteristics--i.e.all choices are considered and the acceptable ones are saved and returned. The grammatical constraints come from dominating slot, which will be the NEW-SLOT field of the attachment point. Thus, in doing our hypothetical realization, we pass the NEW-SLOT field to the filtering function. Since we've seen how grammatical characteristics are checked, we will skip over some code and see how argument characteristics are checked.

```
(defun ARGUMENT-CHARACTERISTICS-ARE-SATISFIED  (the-choice)
  (flet
    ((characteristic-satisfied? (characteristic-and-formal)
       (let ((characteristic (car characteristic-and-formal))
             (formal         (cdr characteristic-and-formal)))
         (let* ((actual (return-rclass-parameter-value formal))
                (test   (consistency-test characteristic))
                (result (funcall test actual)))
           result ))))

    (let ((characteristics (argument-characteristics the-choice)))
       (every #'characteristic-satisfied?
              characteristics))))
```

The argument characteristics of a choice are set of pairs: a one-argument function and a designation of which formal parameter of the realization class that function is to be called with. If the function returns false, then the choice is not acceptable. The function ARGUMENT-CHARACTERISTICS-ARE-SATISFIED is thus relatively simple: the argument characteristics of a choice are implemented as an association list pairing a formal parameter of the realization class with the characteristic whose CONSISTENCY-TEST field contains the unary function we want. We need only look up the actual value of that formal parameter (this uses the RCLASS-PARAMETER-ARGUMENT-LIST, which we bound earlier), and FUNCALL the function. An example of a commonly-used argument characteristic is IDENTICAL-WITH-ROOT, which checks that some argument of the specification to be attached is,

in fact, the same as the one which produced our current phrase (actually, the underlying objects are the same).

Let us return to DISPATCH-AND-CHECK. The other sort of attachment point is a "lowering attachment point." In that case, the definition of the attachment point designates a phrase which will be instantiated and adjoined in at the root of the current phrase. The phrase being adjoined must be a valid realization of the specification so we check the annotated choices in the realization function of this specification until find one that, if taken, would build the same of phrase. If that annotated choice has the right grammatical characteristics (they are consistent with the constraints on our current position), then we have found an acceptable choice. Thus the following function:

```
(defun FILTER-ON-MATCHING-PHRASE (alternative-rclass-choices ap-phrase)
  (flet ((matching-phrase (rclass-choice)
           (and (eq (choice rclass-choice)
                    ap-phrase)
                (grammatical-characteristics-are-satisfied
                  rclass-choice)))
    (remove-if-not #'matching-phrase alternative-rclass-choices))))
```

To summarize: The validity of an attachment point depends on its type and on the possible realizations of the specification being attached in. If the attachment point is splicing, then it will create a new slot in our current tree, so we look through the possible realizations to find one whose grammatical characteristics are compatible with that slot and whose argument characteristics are also satisfied. If the attachment point is lowering, then it will create a new phrase which will be adjoined at the root of the current phrase, so we look for possible realizations that will be the right phrase and also satisfy the grammatical constraints of the current position.

We next look at the case where the attachment function is an attachment class.

```
(defun CHOOSE-AP-AND-ATTACH (attachment-class rspec)
  (multiple-value-bind (ac compatible-choices)
      (choose-from-attachment-class-choices attachment-class rspec)
    (multiple-value-bind (ap position)
        (instance-of-AP-in-present-context (choice ac))
      (attach-at-attachment-point ap rspec position)
      compatible-choices)))
```

```
(defun CHOOSE-FROM-ATTACHMENT-CLASS-CHOICES (attachment-class rspec)
  (let ((alternative-choices (choices attachment-class))
        (usable-acs-and-their-compatible-choices))

    (dolist (ac alternative-choices)
      (let ((ap (choice ac)))
        (if (available-in-present-context ap)
            (let ((compatible-choices
                   (choices-compatible-with-attachment-point ap rspec)))
              (when compatible-choices
                (push (cons ac compatible-choices)
                      usable-acs-and-their-compatible-choices))))))

    (unless usable-acs-and-their-compatible-choices
      (mbug "~a cannot be attached anywhere in the present context~%~
             using the attachment class ~a"
            rspec attachment-class))

    (setq usable-acs-and-their-compatible-choices
          ;; get them back into the order that they appear in
          ;; in the class
          (nreverse usable-acs-and-their-compatible-choices))

    (let ((ac-to-use      (car (car usable-acs-and-their-compatible-choices)))
          (choices-to-use (cdr (car usable-acs-and-their-compatible-
choices))))
      (values ac-to-use choices-to-use))))
```

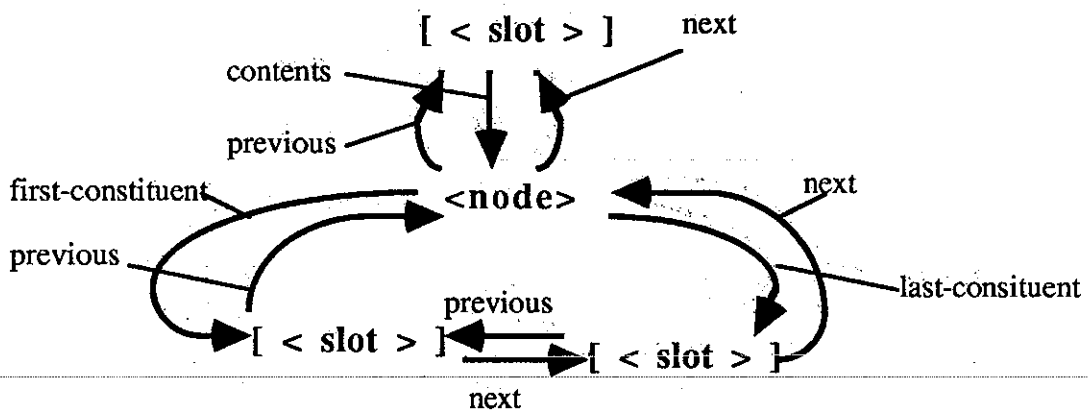
In choosing an attachment point, we iterate over the list of attachment points in the class and call the validating function for each of them, saving the results. It is an error if none of them has any valid choices. Once we are done, we return the first usable attachment point and the choices that go with it. (We collect all usable attachment points because in the actual implementation, we allow the possibility of presenting the user with a menu of them and may add internal criteria for selection in the future.) Once we return to CHOOSE-FROM-ATTACHMENT-CLASS-CHOICES with the attachment point and the compatible choices, we only need to look up the relevant position in the current phrase and call ATTACH-AT-ATTACHMENT-POINT.

```
(defun instance-of-AP-in-present-context (ap)
  (let ((pair (assoc ap (available-aps current-phrasal-root))))
    (values (car pair) ;the attachment-point
            (cdr pair) ;the position
            )))
```

Naturally, Attachment by splicing is different from attachment by lowering, so the first step in doing the actual attachment is to distinguish these cases.

```
(defun ATTACH-AT-ATTACHMENT-POINT (ap rspec position)
  (etypecase ap
    (splicing-attachment-point (attach-by-splicing AP position rspec))
    (lowering-attachment-point (attach-by-lowering AP position rspec)))
  (mapc #'carry-out-action (actions ap)))
```

Splicing Attachment involves just a few simple steps, but, unfortunately, many cases. This is because we have four kinds of links in the Position Path Notation and therefore four different ways of specifying an attachment point. Consider the following picture:



Each slot has a NEXT link and a PREVIOUS link (these are pointer fields in the object). Nodes also have these links, which point to the surface structure above the node. In addition, nodes have links which point down to their constituents; these are the FIRST-CONSTITUENT and LAST-CONSTITUENT links. Surface structure can be attached by breaking any one of these links, so the definition of an attachment point specifies the name of a reference position and a link starting at that position. The splicing code must distinguish these four possibilities.

```
(defun ATTACH-BY-SPLICING  (ap position contents)
  (case (link ap)
    (previous (let ((before (previous position))
                    (after position)
                    (slot-name (new-slot ap)))
                (splice-in before after contents slot-name)))
    (next (let ((before position)
                (after (update position))
                (slot-name (new-slot ap)))
            (splice-in before after contents slot-name)))
    (first (if (nodep position)
               (let ((before position)
                     (after   (first-constituent position))
                     (slot-name (new-slot ap)))
                 (splice-in before after contents slot-name))
               (mbug "unexpected position type")))
    (last (if (nodep position)
              (let ((before (last-constituent position))
                    (after   position)
                    (slot-name (new-slot ap)))
                (splice-in before after contents slot-name))
              (mbug "unexpected position type---A" position)))
    (otherwise (mbug "unexpected type of link---A" (link ap)))))
```

Note that NEXT and PREVIOUS are valid for any type of position, but FIRST and LAST are only valid for nodes. In each case, we simply find the position before the new slot and the position after it (obviously, one of them will be our reference position), and splice in the new slot. That operation is fairly straightforward: we build the new slot, at the same time filling the CONTENTS field with the specification being attached, and modify pointers so that the slot is in the surface structure.

```
(defun SPLICE-IN (before after contents slot-name)
  (let ((slot (make-slot
                :name (name slot-name)
                :labels (list slot-name)
                :visited-status 'new
                :next after
                :contents contents)))
    (cond ((and (slotp before) (slotp after))
           (set-links-for-slot before slot)
           (set-links-for-slot  slot after))

          ((and (slotp before) (nodep after))
           (set-links-for-slot before slot)
           (set-links-for-last-constituent after slot))

          ((and (nodep before) (slotp after))
           (set-links-for-first-constituent before slot)
           (set-links-for-slot  slot after))

          ((and (null before) (nodep after))
           (set-links-for-slot  slot after))

          ((and (nodep before) (null after))
           (set-links-for-slot before slot)))
    slot))
```

In attaching by lowering, the simplifying insight is to remember that we already have a function that will instantiate a phrase, filling slots and initializing tables---INITIALIZE-PHRASE. Therefore, if we make the phrase to be lowered an argument of the phrase we are building, it will be lowered into the correct slot by the normal processing of INSTANTIATE-PHRASE.

Actually, we don't know which argument of the phrase corresponds to the "foot" slot (to use TAG terminology), but we do know that the specification which became the phrase being lowered is identical to the value of some argument of the kernel we are attaching. Therefore, if we replace that argument with the subtree we want to lower , the subtree will end up in the right place.

```
(defun MATRIX-SPEC (s)
  (etypecase  s
    (bundle-specification (matrix-spec (head s)))
    (kernel-specification s)))
```

```
(defun ATTACH-BY-LOWERING (AP position rspec)
  (let* ((matrix-rspec (matrix-spec rspec))
         (rfun         (realization-function matrix-rspec))
         (choice       (select-matching-phrase (choices rfun) (new-phrase
ap)))
         (embedded-rspec  (original-rspec (context-object position))))
    (bind ((rclass-parameter-argument-list
            (create-rclass-parameter-argument-list
              rfun
              (parameters rfun)
              (substitute position
                          embedded-rspec
                          (arguments matrix-rspec)))))
      (let ((phrase (instantiate-phrase (choice choice)
                                        (argument-list choice))))
        (when (bundle-specificationp rspec)
          (bind ((root-node       phrase)
                 (current-phrasal-root (context-object phrase)))
            (process-accessories-&-further-specifications rspec))))))))
```

We call MATRIX-SPEC to go down to find the relevant kernel specification, just as we did when we were validating the attachment point. Then SELECT-MATCHING-PHRASE picks the correct choice, just as Realization would. Finally, we bind the RCLASS-PARAMETER-ARGUMENT-LIST, but we substitute the subtree we want to lower for the correct argument of the kernel. Finally, we call INSTANTIATE-PHRASE. If the specification to be attached was a bundle, we process the accessories and further-specifications, to complete its realization (this usurps normal realization via a bundle-driver).

When the attachment is complete, we return to the function ATTACH-AT-ATTACHMENT-POINT and call CARRY-OUT-ACTIONS. Attachment points can have actions to change the state-vector of the phrasal root. One example is the attachment-point POSSESSIVE, which changes the state to be NO-DETERMINER, since possessive NPs do not have determiners.

```
(defun CARRY-OUT-ACTION (action)
  (dbind (action-type action-value) action
    (ecase action-type
      (:set-state
       (dbind (field value) action-value
         (change-state field value (state current-phrasal-root)))))))
```

Finally, the ATTACH-AT-ATTACHMENT-POINT function finishes and we return to the ATTACH function. In attaching something by splicing, we we can optimize the later Realization by changing the realization-function of that specification to be only those choices that we know are valid. We call this "specializing" the realization-function. In many cases, we'll be able to replace the realization-function with a single-choice. In others, we can replace it with a specialized realization class, created at run time with just the valid choices.

```
(defun SPECIALIZE-REALIZATION-FUNCTION  (rspec usable-choices)
  (let ((kernel-to-specialize
         (rspec-relevant-to-attachment rspec)))

    (cond  ((single-choicep (car usable-choices)))
           ;;single-choice--no need to specialize

           ((and (null (cdr usable-choices))
                 (consp (car usable-choices)))
           ;;only one rclass choice returned and it is
           ;;a parameter of the rclass (returned in a list
            (then ;; Attachment has taken us down to a Single-choice,
               ;; not one or more Rclass-choices.  This means we want
               ;; to revise the kernel directly rather than build
               ;; a specialized class per se.
             (let ((the-single-choice (caar usable-choices))
                   (its-kernel        (cdar usable-choices)))
               (set-realization-function kernel-to-specialize
                                         the-single-choice)
               (set-arguments kernel-to-specialize
                              (arguments its-kernel)))))

           (t  ;; there are several choices, consequently none of them
               ;; can be single choices since they came from an rclass
            (let* ((rclass (realization-function kernel-to-specialize))
                   (specialized-name  (intern
                                       (string-append
                                        "A-SPECIALIZATION-OF-"
                                        (symbol-name (name rclass)))))
                   (specialized-rclass
                     (if (curried-realization-classp rclass)
                         (then (make-curried-realization-class
                                 :name specialized-name
                                 :parameters (parameters rclass)
                                 :choices    usable-choices
                                 :bindings   (bindings rclass)
                                 :reference-realization-class
                                 (reference-realization-class rclass)))
                        (else
                          (make-realization-class
                            :name specialized-name
                            :parameters (parameters rclass)
                            :choices usable-choices)))))

              (set-realization-function kernel-to-specialize
                                        specialized-rclass)

              ))
            )))
```

Attachment is now done, and the bundle-driver regains control. When all further-specifications of the bundle have been attached, the bundle-driver is finished, and therefore Realization of the bundle completes, returning control to Phrase Structure Execution.

3.2 PERIPHERY

3.2.1 Type System

The objects that Mumble-86 manipulates are implemented by means of a custom facility that we refer to as our "type system". The system is comprised of a set of functions, described below, that provide the following features:

A "typed object" abstraction that is independent of the kinds of data-structures in the Lisp being used.

Provisions to define one type as a specialization of another. (In the sense that it has all of the fields of the included type and additional ones as well.)

Automatic, load-time, conversion from the S-expression based representation used in files and text editors (i.e. symbols and lists) to a structured object representation.

Provisions for a definition "language", whereby the internal representation of an object can be an extension or interpretation of the form that is actually typed in.

Cannonical functions for manipulating the objects: creating, printing, accessing or modifying fields, retrieving an object by its name.

An object explicitly representing each type and its properties.

Catalogs of all the objects ever defined, organized by their types.

All objects fall into one of two categories: "temporary" or "permanent". Temporary objects are created by the virtual machine as it runs; in Mumble-86 they are just the positions that make up the surface structure and the specifications that make up the message level. To create them one uses the lower level functions that simply build structures without indexing them, e.g. "Make-node". Temporary objects could be explicitly reclaimed after they have been used, though presently this is just left to the default action of the Lisp garbage collector.

Permanent objects are created by the designers of the system before it is used, stored in files, and loaded in at the beginning of a session. Every permanent object (and many of the temporary ones) has a "name", a Lisp symbol that can be used to refer to the object after it has been converted from its symbolic form in the file where it is defined to its internal form as a "structure" (using the CommonLisp Defstruct facility); the structure will be the value of the symbol at toplevel.[1] Thus we could

[1] When a symbols has been used as the name of several objects, its value will be the object that happened to have been defined most recently given the order in which the files were read in. When there are redundancies like this one must use the <type>-named functions to access the intended structure given its name.

have the following transactions in a lisp listener. (This was taken on a Symbolics Lispmachine.)

```
> sentence
> #<SLOT-LABEL SENTENCE>
> (describe *)
>#<SLOT-LABEL SENTENCE> is a SLOT-LABEL
   POSTPROCESSED?:  T
   NAME:  SENTENCE
   WORD-STREAM-ACTIONS:  (#<WORD-STREAM-ACTION PUNCTUATION>
                          #<WORD-STREAM-ACTION CAPITALIZE-THE-NEXT-WORD>)
   ASSOCIATED-ATTACHMENT-POINTS:  NIL
   GRAMMATICAL-CONSTRAINTS:  (#<GRAMMATICAL-CHARACTERISTIC CLAUSE>)
 #<SLOT-LABEL SENTENCE> is implemented as an ART-Q type array.
It uses %ARRAY-DISPATCH-WORD; it is 6 elements long.
#<SLOT-LABEL SENTENCE>
```

Any object or object type consists simply of a set of named properties, "fields", that take as values other objects. It is an error for two types to be defined to share the same field; when sharing is desired, the type that needs the field should be made to "include" the type that defines it. Given the value restrictions defined for each property as part of the definition of the type, one can organize the set of object types into a lattice according to which types can refer to which others in their fields; this is needed in postprocessing (see below). Mumble-86's lattice is given in the file "objects; postprocessing-order".

Given their uniform design, all object types share the same style of special form for their definition, and are processed in a comparable way. Each type has a lisp macro defined for it with the symbol "define-< type name>", and each such macro has the same pattern of arguments, where "property list" refers to a list of field names (symbols) and expressions.

```
(defmacro  define-<type name>  ( name &rest property-list )  . . . )
```

Being a macro, the interpretation of the expressions that define the field values is entirely up to the defining form; none of them, however, are ever simply evaluated. The particulars are discussed with each of the types in section 3.1.1.

The type "type"

Before a type can be used, it must be defined using the special form "Def-type". This creates and returns an object of type "type", and as a side effect defines a set of functions for the type's manipulation. In many respects Def-type is just the same as CommonLisp's Defstruct, and indeed Defstruct is presently used within the Def-type code to do much of the work, however the differences are significant. Unlike Defstruct, our Def-type creates a first-class object to represent the type itself. This object has fields pointing to useful information about the type, for instance the catalog of all objects of that type ever defined, and can be manipulated just like any other object. Historically, our Def-type facility was developed long before structures

were ever added to Lisp, and served as an abstract datatype that would distance the code of the virtual machine from lisp-specific requirements (e.g. having to implement the objects as list structures).

The type "type" has the following fields:

postprocessed value-restriction: always T

name value-restriction: a symbol

storage-type value-restriction: one of the symbols "permanent" or "temporary"

minimal-construction-function value-restruction: the name of a function

construction-macro value-restriction: the name of the "define-< type name>" macro for this type

type-predicate value-restriction: a function with the name "<type>p"

setters value-restriction: an alist of symbols that name the fields of this type with the corresponding access functions

catalog value-restriction: an object of type "catalog". It will contain a list of all the object of this type presently defined in the system.

properties value-restriction: an alist of the field names with the strings giving their value restrictions.

postprocessing-fn value-restriction: the name of a function.

re-definition-fn value-restriction: the name of a function.

The bulk of these fields are filled by the action of the code for DEF-TYPE and its auxiliaries. What one actually specifies for a type is actually drastically smaller. Below is the definition of the type "node"; notice the use of an included type.

```
(def-type NODE  temporary  position
   (first-constituent  "a slot")
   (last-constituent  "a slot") )
```

The syntax of the DEF-TYPE special form corresponds to the following argument list:

```
(defmacro def-type ( name storage-type included-type
                     &rest list-of-fields-and-their-value-restrictions ) ... )
```

Along with the object that represents the type, the special form DEF-TYPE creates a set of functions for manipulating the objects of that type. For each field, there is an **access function** with the same name, and there is an updating function built out of the field name and the prefix "**set-**". (The update function is a sugaring of the

Commonlisp "setf" and a hook for extensions to the facility that might update cross-indexes when field values are set or changed.)

Also defined is a function for testing whether an object is of that type; its name is the name of the type plus the suffix "**p**", e.g. (nodep obj). To get from a given name (i.e. symbol) to the corresponding object, there is a function defined from the type name and the suffix "**-named**", e.g. (node-label-named 'clause).

Creating, indexing, and "postprocessing" objects

As there is a common protocol to the special forms for defining objects, there is one for processing the definitions and updates to them that are made in the course of a session. The subroutine "create-and-catalog" is used by each of the defining form to do the actual construction (or revision) of the object. This includes making the structure, assigning the values to the fields, and indexing the symbol that has been designated as the object's name.

While it would be an error (and flagged during loading) to use the same name to refer to multiple objects of the same type---not unlike accidentally redefining a function, it is a legitimate and well-defined operation to reuse, "overload", a name in referring to objects of different types. NAME symbols are actually indexed via an association list that pairs objects with their type; this means that one can use the same simple name, say "clause", to designate a node-label, a grammatical-characteristic, and a word. The code of the virtual machine always makes type-specific references, and in the task-level language (i.e. the special forms for defining the various objects for the grammar and message-level) it will always be clear from the context of use which type is intended; in the few cases where more than one type could make sense (e.g. in a stand-alone specification), a preference order has been defined. Since the virtual machine always manipulates real structured objects and never makes only indirect references to them via names, this design seems to make a good deal more sense than insisting on disambiguating the type as part of the name, as in "clause-label". To get the list of all the objects that a symbol has been associated with one retrieves the property **mumble-symbol** from its property list.

Given that the virtual machine exclusively manipulates objects, and that the only conventional option for defining objects is via the symbols and list structure than one can create with a text editor,[2] there must be a facility for converting the S-expression form created by evaling/loading the form in the editor/file into the structured object

[2] With the recent advent of our "interactive demo" facility, it has become possible to think of eventually adding the capacity to the system to define objects on-line via a structure editor and menus. We do not have such a facility a the present time, but it is entirely possible that one may eventually be added.

form, dereferencing all of the symbolic references by replacing names with their corresponding objects.

We talk about this dereferencing activity as **postprocessing**, since it occurs principally at the end of the file-loading sequence after all objects have been defined. The function POSTPROCESS-ENTIRE-SYSTEM is called to do this. It goes through all of the objects in turn, using the lists in the catalogs of each type, and using the type lattice (see above) to dictate the order in which it considers the types (i.e. from the bottom of the reference lattice to the top).

Exactly what happens in order to "postprocess" a symbolic definition is governed by the particular "postprocessing function" that has been defined for that type. Typically the object's fields are scanned for occurrences of object-designating symbols and the corresponding structure looked up using a "<type>-named" function. If there is no structure of the right type already defined for that name, an error occurs; the simplest thing to do to repair the error is to define the needed object on the spot and restart the call to the <type>-named function.

Eventually, when working with Mumble, one might need to redefine an already existing object, i.e. use an editor to change one or more values in the fields of its (symbolic) definition and re-evaluate[3] it. The update will occur transparently with the re-evaluation, since all that will happen is that the field values of the already existing structure will be changed rather than a completely new structure being made; consequently all existing references to the object will be able to remain the same.

Whether postprocessing waits until after all of the object definitions have been loaded or occurs simultaneously with the evaluation of each definition is controlled by the global variable **loading-whole-system**: when this variable is non-nil, postprocessing happens at the end of the loading sequence as the result of an explicit call; when it is nil it happens as part of the call to Create-and-catalog. The function **postprocess** is used when the operation is such an embedded call.

3.2.2 Tracker

The Tracker subsystem provides the user with information about what Mumble is doing. It is based on a set of "landmarks" which are placed throughout the code. Each landmark has its own associated schema which governs its actions (i.e what it will print, what form it will print it, what objects it will highlight). There is an interface for turning landmarks on and off individually, allowing the user to concentrate attention on particular aspects of Mumble's execution. The Tracking System has three major operations:

[3] Since object definitions are simply schematic data, there is no appreciable gain to be had from compiling them rather than simply evaluating them.

1) Presenting tracing information: The tracker prints text to the "tracker" pane in the Mumble constraint frame for moment by moment status information and also to an editor buffer so that it may be reviewed later.
2) Highlighting objects: This feature controls the "you are here" pointers in other windows of the browser, e.g. the message, surface structure, and realization class windows.
3) Controlling the progress of the program: This feature offers the user stepping mode, pausing at a predefined point (defined in terms of a landmark), and the option for entry into a break loop.

Overview

The tracker's presentation of Mumble's activity can be organized in two ways: The usual one is as a continuous stream of status information that is produced as long as the tracker is "on". The user can designate which individual landmarks are to be included in this stream (such landmarks are "on") and thereby control the level of detail that they see. A complementary organization is to focus the presentation around specific objects, e.g. the execution of a particular realization class, or while traversing a given subtree. This "object-based" mode is particularly useful when debugging; the information stream is presented between the moment that the object is reached and when it is left (such moments will be particular to the kind of object being tracked).

The implementation of landmarks is done by a common interpreter processing the data in each landmark's schematic definition. We can think of that interpreter as equivalent to having defined each landmark as a lisp function with the following structure:

```
(update-major-mode)
 (cond-every
   ((or in-stepping-mode (this-landmark-is-on? self))
    (mapc #'(lambda (stream) (format stream "..." obj arg1 arg2))
        various-streams)
    (mapc #'(lambda (window) (send-if-exposed window :highlight obj))
        various-windows))

   (in-stepping-mode
     ;; don't continue until the user clicks on the mouse
    (do () ((char= (send *perspicuous-place* :any-tyi)
              #\mouse-left))))

   ((or (is-this-landmark-used-for-object-tracking? obj)
      (is-this-landmark-used-for-landmark-tracking? obj))
        (turn-on-the-landmarks-that-the-user-previously-told-us-to obj)))
```

The "major mode" will usually be the process ("Attachment", "Realization", or "Phrase Structure Execution") in which the landmark appears. "In-Stepping-mode" is a global flag. The variable "obj" refers to the distinguished object that a landmark will typically designate, e.g. when announcing the start of realization the designated object is likely to be the realization class. This object may be highlighted if it appears in a browser pane

We will organize the rest of this section in terms of the Tracker's three interfaces: the interface between Mumble and the Tracker, where calls to the "landmark" function occur at specific points in the code for Mumble's interpreters; the interface between the user and the Tracker, which lets users specify which landmarks they would like to be active; and the interface with the programmer, who defines the schemas for the landmarks and thereby what will happen at them (e.g. the wording of the trace message that is printed out).

User Interface

Currently, the user-interface consists of two items in the Mumble command menu. These items are as follows:

[Stepping] This controls global state variables of the Tracker, turning stepping mode on and off, and, when in stepping mode and paused at a landmark, telling the Tracker to continue to the next landmark.

[Tracker] This pops up a subsidiary menu of the following items:

(1) Turn Tracking System On.

(2) Turn Tracking System Off. These two modify a global variable that short-circuits the execution of landmarks, bypassing all Tracker code and returning immediately to Mumble.

(3) Clear current tracking activity, which is a quick way to turn all the landmarks off. This is not quite the same as turning the Tracking system off, since only the visible sign of the landmarks is turned off---they don't print, pause, highlight or whatever. They do, however, execute any "must-code" (code that is supplied in a landmark definition and which the programmer wants executed whether the landmark is on or off). They also record information in internal arrays, for use in later runs.

(4) Change Tracking Activity, which pops up a menu of the landmarks so that they can individually be turned on or off.

Program Interface

We conceive of the program interface as a call to the "landmark" function, inserted in many places throughout the Mumble code. The landmark function has two syntaxes:

```
(landmark 'name-of-landmark &rest args)
(landmark '({:before :after} name-of-landmark) &rest args)
```

A schema specific to the named landmark is looked up by the landmark function and applied to the arguments. The distinguished object (by convention the first argument) is highlighted in an appropriate pane; as part of its internal data structure, the named landmark knows what pane is appropriate. The specific code for this landmark is a basically a FORMAT string, which, together with these arguments, will determine what is printed to the various streams.

The second syntax is not very different from the first, except that it has a structured name, i.e. a list consisting of a keyword (one of the two choices in braces) and a symbol; it is used to designate landmarks for object-based tracking. An "enclosing-object" can be included to either calling form (as its second argument), to indicate what the distinguished object is part of and thereby disambiguate it from other instances of the same object. (E.g. when the distinguished object is a label the enclosing object would be the position it appears in.)

Programmer Interface

The programmer interface allows a Mumble programmer to add a new landmark, or add a new datatype for object-based tracking. We have implemented

several special forms to help the programmer to make these additions. The first is Define-trackable-datatype for object-based tracking.

```
name-of-datatype ::= {rspec, rclass, word-instance, choice, annotated-choice, label ...}
landmark-name ::= symbol, ({:before, :after} symbol)
(define-trackable-datatype name-of-datatype
  (:interval landmark-name [landmark-name])
  (:relevant-flags \{landmark-name\}+ , :all)
  (:default-flags  \{landmark-name\}+ , :all, :same)
  (:notes [:map-to-number] {:permanent, :ephemeral}))
```

This is a regular-expression-like notation, based on association lists for the general look of the parenthesizing. The function of each clause is as follows:

:INTERVAL gives the pair of landmarks that define the interval before and after the processing (and hence the tracking) of an object of this datatype. If only one is given, (say LANDMARK1), then it implies the pair of landmarks '(before LANDMARK1) and '(after LANDMARK1). If two are given, then they are both landmark-names.

:RELEVANT-FLAGs is a list of landmark-names (since there is presently one flag controlling each landmark, the two are somewhat synonymous), and is used in constructing the menu for the user interface (see above).

:DEFAULT-FLAGS is a subset of the relevant flags, and all of these are implicitly turned on in the given interval. Either of these clauses can instead contain the keyword :all meaning that the menu of all possible flags should be used.

:NOTES contains particular informtion about the datatype. :MAP-TO-NUMBER tells the system that it will not encounter this specific object (the one indicated via the user interface), but should look in its history array of what happened in the ongoing run and use that to map the indicated object into a number, and them make the landmarks sensitive to the objects with a comparable numerical index. This would be used for slots, for example. The other two keywords are mutually exclusive and tell the system whether it should clear this trigger at the end of a run (they are always cleared if the datatype is ephemeral); for instance, realization classes are permanent, but specifications are ephemeral.

The first three clauses, taken together, may be omitted, so that a programmer can tell the tracking system about a datatype that a user might click on (and that can be encountered during the run), but which does not have beginning and ending landmarks. Since such datatypes could not be used to specify object-based tracking, the relevant flags must be :all; the :notes clause is still possible.

The other principal special form is "define-landmark".

```
name-of-landmark ::= symbol
arglist ::= {symbol}*
pane ::= {rspec-display, rclass-display, surface-structure-display }

(define-landmark name-of-landmark (arglist)
  "presentable string"
  (:major-mode "short string")
  (:applicable-datatypes {name-of-datatype}*)
  (:message format-string . forms)
  (:before  format-string . forms)
  (:after   format-string . forms)
  (:highlight pane)
  (:must-code <arbitrary lisp code>)
  (:maybe-code <arbitrary lisp code>))
```

The name of a landmark is just a symbol. If either a :before or :after clause appears in the body, then we are defining an object-based landmark, and the name will be combined with the appropriate keyword when the landmark is used. Since such a pair of landmarks might be entirely different except for their name (they might even have different distinguished objects---consider before and after realization: before, we have a specification and afterwords we have node or a word), we need separate definitions for them. Next, we have an argument list, which is just a simple list of symbols. Since each landmark will be referred to exactly once in the Mumble system, there is no need to have multiple calling forms, such as allowing "&optional" arguments.

The argument list is a list of symbols which is mapped onto the "&rest" arguments in the call to the landmark function. The symbols distinguished-object and enclosing-object will always be bound to the first two arguments. The argument list is primarily useful to help the programmer remember how the landmark is called. The format-string defines what is to be printed on the various streams. When we just want to define one landmark (not a pair of before and after landmarks), we use the :message clause, otherwise one of the other two. (That is, the clauses :message, :before and :after are mutually exclusive.)

The highlight clause is optional, and defines what pane, if any, to send a highlighting message to. The arguments with the message are always the distinguished-object and the enclosing-object. :Applicable-datatypes are the datatypes we expect the distinguished object to be. Typically, there will only be one element in the list, but some landmarks may have more. This helps us do the winnowing of landmarks by datatypes that was mentioned in the discussion of the user interface. The :major-mode clause is used to update the small pane on the Mumble frame that tells the user what is going on overall. Again, this is something like "Phrase Structure Execution," or "Attachment." It has to be short enough to fit

in the pane. It defaults to "Running." Finally, the "presentable string" is optional, and is something that is more suitable to be put into a menu than the name of the landmark (which since it must be a symbol is typically long and hyphenated). Otherwise, the menue form is created by lowercasing the print-name of the landmark symbol and replacing hyphens with spaces.

There are two clauses that take arbitrary code. This code is executed by the Tracker at the moment in Mumble's execution when the landmark is called. This allows experimental modification of Mumble's execution without having to touch any of the actual code of the virtual machine. There are two sorts of arbitrary code: "maybe" and "must." The former is executed only when the landmark is "on," while the latter is always executed whenever the landmark is encountered (assuming the Tracker is turned on).

Landmark forms and functions:

[LANDMARK] This function is called by Mumble during its execution and defines the run-time functionality of the Tracker. Anything the Tracker does while Mumble is running happens during the execution of this function. If each landmark is like a function, then this function is analogous to FUNCALL, in that it will run the landmark.

[DEFINE-LANDMARK] This form names a landmark that will be encountered during Mumble's execution and says what should happen when that call to the LANDMARK function occurs. If landmarks are like functions, then this form is analogous to DEFUN, because it is part of the programmer interface.

[DEFINE-TRACKABLE-DATATYPE] This form is also part of the programmer interface, but it defines one of the datatypes that might be encountered by a landmark. It exists mostly to serve the user interface for object-based tracking, while DEFINE-LANDMARK serves the program interface.

[BEGIN-TRACKER-RUN and END-TRACKER-RUN] These functions supplement the landmark interface between Mumble and the Tracker. They are Tracker functions which Mumble must call before and after a run, respectively, so that the Tracker can set things up and clean up afterwards.

[STEPPING?] This form returns the current state of the Tracker with respect to stepping: either true or false. It can be SETF'd so that the user can also activate or deactivate stepping mode.

[TRACK-TYPE] This is a unary function that gives the Tracker the `type' of a Mumble object. The type should be one of the set defined as `trackable-datatypes.' Essentially, this function is just a connection between Mumble's type system and the trackable datatypes.

The following are macros that act like global variables (they return a value and the value can be set via SETF). They affect the global state of the Tracker, rather than the functioning of a particular landmark. The first two are boolean and say where the printed output should go. The third is tri-valued (:ignore, :warn, :error) which say how the Tracker should deal with errors. The fourth should evaluate to a stream where errors are to be reported in the case that ERROR-ACTION is :warn. Naturally, if ERROR-ACTION is :ignore, THE-ERROR-STREAM is meaningless, and if it is :error, then the break-loop and error message occur in the Lisp Listener, regardless of the value of THE-ERROR-STREAM. The last item says what pane the "major mode" information, which describes what process Mumble is in, is to go.

OUTPUT-GOES-TO-EDITOR-BUFFER?

OUTPUT-GOES-TO-STANDARD-OUTPUT?

ERROR-ACTION

THE-ERROR-STREAM

PROCESS-PANE

[TRACK-FORMAT] This function is the Tracker's equivalent of the Lisp FORMAT function. It is available in case anyone wants to print to the streams that the Tracker is using.

[*TRACKER-COMMANDS*] This list is available so that people running Mumble without a browser can use some sort of menu-choose to control the tracker. It is a list of menu-items in the correct format for tv:menu-choose or mg-lisp:menu-choose.

3.2.3 BROWSER

The Core Mumble will run without anything more than a Lisp Listener: the text is printed to the screen, so Mumble is rather like a slow format function. But, since we have Lisp Machines with bit-mapped graphics, fonts, and mice, we can create a better user-interface to Mumble. We can make common commands available via menus; we can lay out the screen for special purposes, such as development work versus demonstrations; and we can create display windows that show us what Mumble is doing at each moment. All this is what we call "the browser." To turn the browser off, so that you are only interacting with the Core Mumble, simply go to the bootstrap file (see the section on the Loader) and set the variable *WINDOW-CODE?* to nil.

The Constraint Frame

The organizing unit for all this is the *MUMBLE-FRAME*, which is just a lisp machine constraint frame. All the configurations include a lisp listener, which is the process that actually runs Mumble. All the configurations include a "text output" window, which is just an instance of tv:window, and shows Mumble's output in some pleasing font. All the configurations have a command menu, which has options to do the following:

--Change configuration
--Run a demonstration message (such as "Fluffy")
--Control the Tracker (see the section on the Tracker)
--Turn on/off Stepping mode (allowing one to single-step Mumble's execution)
--Interrupt Mumble (making it stop in a place from which it's easy to continue)

Some configurations include an editor, since it's usually convenient to be able to have both the source code and the lisp listener on the screen at once when you're doing development.

The other panes, when they exist, are all essentially read-only display windows that tell something about what Mumble is looking at or doing. One simple one is the Process Pane, which does nothing more than print out in a big font the "process" that Mumble is in: Realization, Phrase Structure Execution, or Attachment.

Line Buffer Geometry

The other windows[4] all exist to pretty-print Mumble data structures (such as realization classes or the input message) and "highlight" them (by putting a blinker around them) when they are relevant to Mumble's execution. These have been our design goals:

to pretty-print objects into windows using multiple proportional-spacing fonts
to make the printed representation (p.r.) scrollable
to make the p.r. mouse-sensitive
to make the p.r. highlightable by Mumble

To attain these goals, we built two tightly integrated chunks of code: The Display Engine and the line-buffer-geometry window flavor. The Display Engine provides functions for use in pretty-printing routines and builds representations of the lines that were printed and the mouse-sensitive items that were printed. A mouse-sensitive item covers a rectangle of text (the p.r. of some object) and can be either smaller or larger than a line. The line-buffer-geometry window flavor, which understands our representations of lines of text and mouse sensitive items can track the mouse over the contents of the window. It also can scroll the window and highlight mouse sensitive items by putting a blinker around the p.r. of the object.

[4]Except the windows that make up the Interactive Demonstration facility, documented in Section 7.

The Display Engine

Most Lisp programmers do output, especially pretty-printed output, using the FORMAT function, because it's simple and powerful. Nevertheless, it's relatively straightforward to write a pretty-printer for some object using only the Common Lisp functions PRINC, TERPRI, and FRESH-LINE, and a global variable for keeping track of the indentation. In a pinch, if FORMATprovides some feature that you want to use, you can use (format nil ...) and PRINC the resulting string. Therefore, for our Display Engine, we will provide analogs of the Common Lisp functions, as follows[5]:

[DPRINC] This is the Display Engine's version of PRINC. Like PRINC, it does not do a carriage return after printing its argument. Unlike PRINC, it (1) only takes strings as arguments, and (2) takes a second argument, the FONT that the string should be printed in.

[PP-NEWLINE] This is the Display Engine's version of TERPRI. This function takes no other arguments.

[PP-FRESHLINE] This is the Display Engine's version of FRESH-LINE This function takes no arguments.

[SPACE-OVER] This function prints some number of spaces. The only argument is an integer, the number of spaces. It is optional and defaults to 2. Why not just do a (DPRINC " ") Well, first of all, DPRINC does a fair amount of work, so SPACE-OVER can be a lot more efficient. Secondly, the amount of whitespace with the DPRINC method depends on what font the space is in. Even if you let the spacing always be in the default font, that may mean unnecessary work for the machine. For instance, you put two spaces between two things printed in bold. The machine has to do two font changes in there--before and after printing the spaces. Consequently, we have invented "canonical spaces" which are some fixed number of pixels wide, so we can simply increment the cursor position and not bother with fonts. Having a canonical space also makes it easier to line things up.

[*LEFT-MARGIN*] A global variable that keeps track of the current level of indentation, measured in "canonical spaces." Both PP-NEWLINE and PP-FRESHLINE observe the value of this variable, so calling either gets you to the left margin, whatever that is, not necessarily to the left edge of the window.

[5]Warning: We might have simply prefixed each function name with, for example, "pp-," but we've renamed some of the functions because of our own sense of aesthetics.

[INDENT] This is just a macro which increments *LEFT-MARGIN* for the scope of its body. It takes an integer, which is the number of canonical spaces to increment the *left-margin* by, and then any number of forms in the body.

[*NORMAL-FONT*, *BOLD-FONT*, *ITALIC-FONT*, and *BOLD-ITALIC-FONT*] are just global variables that are bound to members of a nice proportional font family. They make convenient arguments to DPRINC, though you can certainly call it with any font you want.

[MOUSE-SENSITIVE-OBJECT] This function is how the Display Engine is told what objects and printed representations are to be mouse sensitive. Its arguments are as follows:

```
(object pretty-printing-routine &key
                        (font *normal-font*)
                        differentiator
                        enclosing-object
                        save-state?)
```

MOUSE-SENSITIVE-OBJECT pretty-prints OBJECT to a line-buffer-geometry window (actually, to buffers which will later be painted on the window) using PRETTY-PRINTING-ROUTINE. The PRETTY-PRINTING-ROUTINE must be a two-argument function, the first argument is the object and the second is the font. You can specify the font to be given to the pretty-printer as a keyword argument (for example, :font FONTS:CPTFONT).

If the object is to be highlightable by Mumble and is printed in more than one place in the window, the various instances (printed representations) must be differentiated. The differentiator is specified by a keyword argument and can be any object. It defaults to nil, which is the easiest thing to do if the object is to be printed in only one place and therefore needs no differentiator.

If the printing is never to be highlighted, give a differentiator of :NO-HIGHLIGHT.

If the object is to be replaceable by the Interactive Demo facility, then the enclosing object must be given so that the field of it which contains the printed object can be found and modified. Again, this object is given as a keyword argument. For example, suppose we're printing specification BAR which is one of the arguments of specification LANDMARK1, and we want the Interactive Demo to be able to modify the fields of LANDMARK1. All we need to do is give LANDMARK1 as the enclosing object when we print BAR. When, later, a user clicks on the printed representation of BAR and says 'replace this', we know to modify LANDMARK1.

Lastly, if the printed representation of something may be replaced with another printed representation, as can occur in realization, we will want to save the

state of the display engine so that we can easily re-print the object. To do that, specify the keyword argument :SAVE-STATE? with the value T.

[FINISH-CURRENT-LINE] This takes care of various internal issues when we're done printing on a line. Typically, you won't need to worry about this since it's automatically called by PP-NEWLINE. However, there may be times when you don't want to start a new line (which PP-NEWLINE will do), such as when you're done printing everything. In that case, you can simply call this function directly.[6]

[INITIALIZE-DISPLAY-ENGINE] This takes one argument, the line-buffer-geometry window you're printing to, and just initializes the state variables of the display engine (like starting *left-margin* off at zero). Call this function first.

[FINALIZE-DISPLAY-ENGINE] is the matching function. It's called to finish up things (such as creating the data structures for mouse-tracking) when you're done printing to a window. Takes the window as its only argument.

Using these functions isn't very hard. The general idea is just to

1. Make a line-buffer-geometry window
2. Call INITIALIZE-DISPLAY-ENGINE with that window
3. Traverse your data structure or whatever, making calls to DPRINC, PP-NEWLINE, SPACE-OVER, PP-FRESHLINE, and INDENT.
4. Call FINALIZE-DISPLAY-ENGINE with your window.

Now, things get a little more complicated when you want mouse sensitivity. The problem is that the mouse-sensitive items must be nested rectangles of text. The Display Engine's MOUSE-SENSITIVE-OBJECT function figures out the corners of your mouse-sensitive region by recording where its 'cursor' is before it calls your pretty-printing function and where it is afterwards. If you move the cursor (e.g. by calling PP-NEWLINE or PP-FRESHLINE) before starting your text or after ending it, you will probably find that the resulting mouse-sensitive region is the wrong shape. The solution is simply to provide a pretty-printing function that just prints a rectangle of text, without a prelude or postlude of blank lines.

[6]The only case that I've seen where this function was called by the 'user' was after the very last line to be printed. Perhaps we should put a call to this function in FINALIZE-DISPLAY-ENGINE.

This means it's the responsibility of the function which calls MOUSE-SENSITIVE-OBJECT to put the cursor in the correct place before making the call.[7] Here's an example chunk of code for printing the message:

```
(defun initialize-message-window-and-display-message (rspec)
  (let ((w   *message-display-window*))
    (when (send w :exposed-p)
      (send w :reinitialize)
      (send w :set-message rspec)
      (initialize-display-engine w)
      (mouse-sensitive-object rspec 'pp-message)
      (finish-current-line)
      (finalize-display-engine w))))

(defun pp-message (m &optional font)
  (etypecase m
    (kernel-specification  (pp-kernel m))
    (bundle-specification  (pp-bundle m))))

(defun pp-kernel (k &optional font)
  "Kernel specification: <realization-function>
                         <arguments>"
  (mouse-sensitive-object
    (realization-function k) 'pp-object :differentiator k)
  (space-over)
  ;; the following prints something like "S13"
  (pp-spec k)
  (indent 3
    (dolist (arg (arguments k))
      (pp-newline)
      (typecase arg
        (word-stream-item
         (mouse-sensitive-object arg 'pp-object
                                 :font *bold-italic-font*
                                 :differentiator k
                                 :enclosing-object k))
        (specification
         (mouse-sensitive-object arg 'pp-message :enclosing-object k))
        (otherwise
         (error "unexpected type of argument to ~s:  ~s" k arg))))))

(defun pp-object (object font)
  (dprinc (string-downcase (short-description object)) font))
```

[7] Because of the tighter choreography of the pretty-printing functions, I find that I rarely use PP-FRESHLINE, just because I always know whether I'm on a new line or not. Small loss.

4. A WALK THROUGH AN EXAMPLE

Contents

In this section we present a detailed walkthough of Mumble generating a sentence from a stand-alone specification in the input specification language. We describe each step and show the actual Mumble objects used in the realization of *"Fluffy is chasing a little mouse"* starting from a standalone "demo". (Alternative starting points are discussed in Section 6.)

4.1 Starting up

The stand-alone interface begins with the definition of a demo, as shown in Figure 4.1. The function CREATE-MESSAGE turns these symbols and lists into actual objects in Mumble's input specification language. (See discussion in Section 2.2).

```
(def-demo-rspec  FLUFFY
   "Fluffy is chasing a little mouse"
   "a standard example"
   (discourse-unit  <R1>
     :head (general-clause  <R2>
             :head (chase  <R3>
                     (general-np  <R4>
                       :head (np-proper-name "Fluffy"  <R5>
                       :accessories (:number singular
                                     :determiner-policy no-determiner))
                     (general-np  <R6>
                       :head (np-common-noun "mouse")  <R7>
                       :accessories (:number singular
                                     :determiner-policy kind)
                       :further-specifications
                         ((:specification
                             (predication_to-be *self*
                                                (adjective "little"))
                           :attachment-function restrictive-modifier))
                        )))
            :accessories (:tense-modal present
                          :progressive
                          :unmarked)) ))
```

FIGURE 4.1

Realization in Mumble is always within a linguistic context. Initially, when Mumble begins, there is, of course, no established context and a slot labeled TURN is created and the whole specification becomes its contents. The top level bundle DISCOURSE-UNIT takes care of building the initial context. Figure 4.2[1] shows the bundle type DISCOURSE-UNIT and its associated bundle driver.

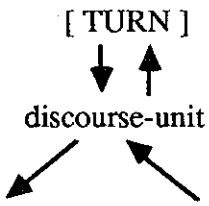
```
(define-bundle-type DISCOURSE-UNIT
                    discourse-unit-bundle-driver )

(defun DISCOURSE-UNIT-BUNDLE-DRIVER (bundle)
  (let* ((phrase (phrase-named 'first-sentence-of-discourse-unit))
         (parameter (car (parameters-to-phrase phrase))))
    (let-with-dynamic-extent
          ((phrase-parameter-argument-list
             (list (cons parameter (head bundle)))))
      (setq pending-rspecs (further-specifications bundle))
      (build-rooted-phrase (definition phrase)))))
```

FIGURE 4.2

This procedure uses the phrase FIRST-SENTENCE-OF-DISCOURSE-UNIT to build the surface structure shown in Figure 4.3. It places the head of the bundle in the slot labeled SENTENCE and any further specifications on a list of PENDING-RSPECS (in this case there are none).

```
(define-phrase FIRST-SENTENCE-OF-DISCOURSE-UNIT (R)
  (discourse-unit
    sentence  R))
```



< R2: (general-clause
:head (chase ...>

FIGURE 4.3

At this point *phrase-structure-exectution* (PSE) takes over. (PSE is discussed in detail in section 2.5.) Beginning with TURN, PSE traverses the tree. When the traversal reaches the label SENTENCE, it first executes the associated "inital" word

[1] The syntax of the special forms shown throughout this example is explained in detail in section 3.1.1.

stream actions (see Figure 4.4), in this case dropping a flag (called a "blip") into the word stream to capitalize the first word of the sentence.

```
(define-slot-label SENTENCE
  grammatical-constraints (clause)
  word-stream-actions ((punctuation final period)
                       (capitalize-the-next-word initial)))
```

FIGURE 4.4

Next, since the contents of the slot is an unrealized specification, the process *realization* takes over. When the specification is a bundle, as in this case, the driver of the bundle defines the order of the realization of its parts. Figure 4.5 shows the bundle type and driver for GENERAL-CLAUSE.

```
(define-bundle-type GENERAL-CLAUSE
  clausal-bundle-driver
  accessory-list (:question
                  :tense-modal
                  :perfect
                  :progressive
                  :negate
           ;;the following are passive accessories (unordered)
                  :wh
                  :command
                  :purpose-clause-object
                  :given
                  :unmarked))

(defun CLAUSAL-BUNDLE-DRIVER (bundle)
  (landmark 'realizing-bundle-specification bundle)
  (let ((result (realize-kernel-specification (head bundle))))
    (if (nodep result)
        (then (set-backpointer-of-root result bundle)
              (let*-with-dynamic-extent
                ((root-node        result)
                 (current-phrasal-root (context-object root-node)))
              (process-accessories-&-further-specifications bundle)
              root-node))
        result)))
```

FIGURE 4.5

There are three major steps in this bundle driver: 1) realize the head of the bundle, 2) process the accessories, and 3) realize any further specifications. Not until all three steps are finished is the result knit into the SENTENCE slot.

4.2 Realizing the head

The head of this bundle (<R3> in Figure 4.1) is a kernel specification. The first element of the list, CHASE, is the realization function; the other elements, <R4> and <R6> are its arguments. CHASE, shown in Figure 4.6, is a curried-realization-class,

that is, it is a function with two arguments that "curries" in an addition argument (in this case the verb "chase") and calls another function, in this case the realization-class TRANSITIVE-VERB_TWO-EXPLICIT-ARGS (shown in Figure 4.7).

```
(define-curried-realization-class CHASE (agent patient)
  Transitive-verb_two-explicit-args
  ((verb "chase")))
```

FIGURE 4.6

A realization class is a predefined set of alternatives annotated by the characteritsics that distinguish them. In the class shown in Figure 4.7, each choice represents a possible phrasal realization of a transitive verb with two explicit arguments, the agent and the patient. GRAMMATICAL-CHARACTERISTICS, such as CLAUSE in choice #1 or RELATIVE-CLAUSE in choice #2, are compared with the grammatical constraints on the labels of the current position. In this case, the current position is SENTENCE (see surface structure representation in Figure 4.3 and label definition in Figure 4.4 above) which has the grammatical constraint CLAUSE. Choices 1, 9, 10 and 11 are acceptable using this filter. (Note, number of the choices are only used here for reference; they are not part of the actual object)

REQUIRED-ACCESSORIES are compared with the accessories on the bundle being realized (see Figure 4.1 above). This further filters the acceptable choices to just #1. (If more than one choice had been acceptable, it would have chosen the one that came first in the class.)

```
(define-realization-class TRANSITIVE-VERB_TWO-EXPLICIT-ARGS (verb agent
                                                             patient)

      1  ((SVO  agent verb patient)
          :grammatical-characteristics (clause)
          :required-accessories (:unmarked))

      2  ((SVO-subj-rel  agent (agent :trace) verb patient)
          :grammatical-characteristics (relative-clause)
          :argument-characteristics (identical-with-root agent))

      3  ((Svo-obj-rel  patient  agent  verb  (patient :trace))
          :grammatical-characteristics (relative-clause)
          :argument-characteristics (identical-with-root patient))

      4  ((SVO-for-inf   agent verb patient)
          :grammatical-characteristics (for-infinitive))

      5  ((SVO-for-inf    (agent :trace) verb patient)
          :grammatical-characteristics (for-infinitive)
          :required-accessories (:purpose-clause-object agent))

      6  ((SVO-for-inf    (agent :trace) verb patient)
          :grammatical-characteristics (for-infinitive)
          :argument-characteristics (available agent))

      7  ((SVO-for-inf    agent verb (patient :trace))
          :grammatical-characteristics (for-infinitive)
          :required-accessories (:purpose-clause-object patient))

      8  ((SVO-for-inf    (agent :trace)  verb (patient :trace))
          :grammatical-characteristics (for-infinitive)
          :required-accessories (:purpose-clause-object patient)
          :argument-characteristics (available agent))

      9  ((SVO-subj-whq  agent (agent :trace) verb patient)
          :grammatical-characteristics (clause)
          :required-accessories (:wh agent))

      10   ((SVO-obj-whq   patient   agent   verb   (patient :trace))

           :grammatical-characteristics (clause)
           :required-accessories (:wh patient))

      11  ((SVO (agent :trace) verb patient)
           :grammatical-characteristics (clause)
           :required-accessories (:command))
          )
```

FIGURE 4.7

4.3 Building the phrase.

The definition of the phrase just chosen is shown in Figure 4.8. The next step of the process is to instantiate the phrase. The arguments to the kernel being realized are bound to the parameters of the phrase, which defines their position in the surface

structure. Note that the arguments are merely placed in the structure. They are not realized until their position is reached by phrase structure execution.

```
(define-phrase SVO (s v o)
  (clause :set-state (:aux-state initial)
    subject s  :additional-labels (nominative)
    predicate (vp
                verb v
                direct-object o  :additional-labels (objective))
  ))
```

FIGURE 4.8

Labels on the phrase not only indicate grammatical constraints and word stream actions, but they also have attachment points which define where more phrases may be added. Additional labels on a phrase, such as nominative and objective, are accessed during morphology to govern such things as case markings on pronouns. (NOTE: additional labels are not shown in the surface structure diagrams.)

When a phrase is instantiated, the root of that phrase is designated as an object of type PHRASAL-ROOT. It keeps a list of the attachment points associated with the positions in the phrase and the STATE of the phrase. The STATE keeps information needed by more than one constituent of the phrase, such as number in noun phrases (this need is reflected in English only in such phrases as "these dogs") and state of the verb group (see section 2.4.1). In this example, the phrasal root is CLAUSE and its state is set to INITIAL.

This completes the realization of the head of the bundle; the result is shown in Figure 4.9.

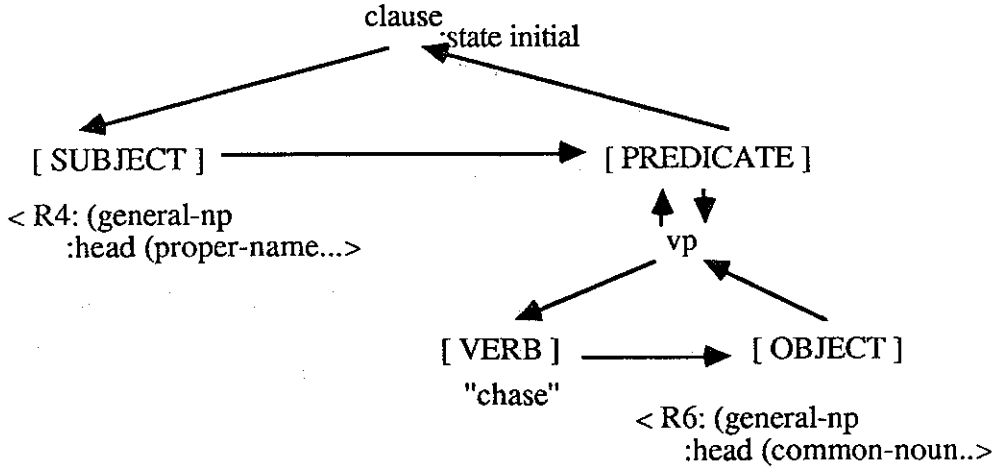


FIGURE 4.9

4.4 Processing the accessories

The next step in the realization of the bundle is to process the accessories. In this example, the first is TENSE-MODAL, whose type and processing function are shown in Figure 4.10. This accessory takes a value which may be either the tense-marker PAST or PRESENT or a modal verb (*will, can, could, would, should,* etc.). In this case the value is the tense-marker PRESENT.

```
(define-accessory-type :TENSE-MODAL
  (past present (modal)))

(defun PROCESS-TENSE-MODAL-ACCESSORY (value)
  (let* ((ap (return-tense-modal-attachment-point))
         (active-ap (assoc ap (available-aps current-phrasal-root)))
         (position (cdr active-ap))
         (contents
           (cond ((or (eq value (accessory-value-named 'past))
                      (eq value (accessory-value-named 'present)))
                  (tense-marker-named (name value)))
                 ((wordp value) value)
                 (t (mbug "unexpected contents of tense-modal ~a" value)))))
    (let ((new-position (attach-by-splicing ap position contents)))
      (delete! ap (available-aps current-phrasal-root))
      (push (attachment-point-for-next-aux new-position contents)
            (available-aps current-phrasal-root))
      (push (cons 'tense-marker new-position)
            (position-table current-phrasal-root))
      )))
```

FIGURE 4.10

The processing function gets the attachment point TENSE-MODAL (whose definition is shown in Figure 4.11) from the active attachment point list. (This attachment point was activated by the label subject; see Figure 4.17 below.)

```
(define-splicing-attachment-point TENSE-MODAL
  reference-labels (subject)
  link (next)
  new-slot (tense-modal))
```

FIGURE 4.11

It then splices a new slot after the subject, puts the label TENSE-MODAL on the position and the value of the accessory as its contents. This label activates other attachment points for another auxiliary or negation (see Figure 4.12).

```
(define-tense-marker PRESENT )

(define-slot-label TENSE-MODAL
  associated-attachment-point (negation next-aux))
```

FIGURE 4.12

The next accessory to be processed is the PROGRESSIVE, whose processing function is shown in Figure 4.13. This accessory works in the same manner as TENSE-MODAL, splicing a new position in the surface structure, labeled BE+ING with contents "be".

```
(defun PROCESS-PROGRESSIVE-ACCESSORY ()
  (let* ((ap (splicing-attachment-point-named 'next-aux))
         (ap-set (assoc ap (available-aps current-phrasal-root)))
         (position (cdr ap-set))
         (contents (word-named 'be)))
    (set-new-slot ap (label-named 'be+ing))
    (attach-by-splicing ap position contents)
    (delete! ap (available-aps current-phrasal-root))
    (push (cons 'be+ing position)
          (position-table current-phrasal-root))
    (if (eq (state-value ':aux-state (state current-phrasal-root))
            'prepose-aux)
        (change-state ':aux-state 'initial
                      (state current-phrasal-root)))
  ))
```

FIGURE 4.13

This completes the realization of the bundle R2; the result is knit into the surface strucure as shown in Figure 4.14 and control returns to the phrase structure execution process. (NOTE: arrows in bold in the surface structure diagrams represent arcs already traversed by PSE.)

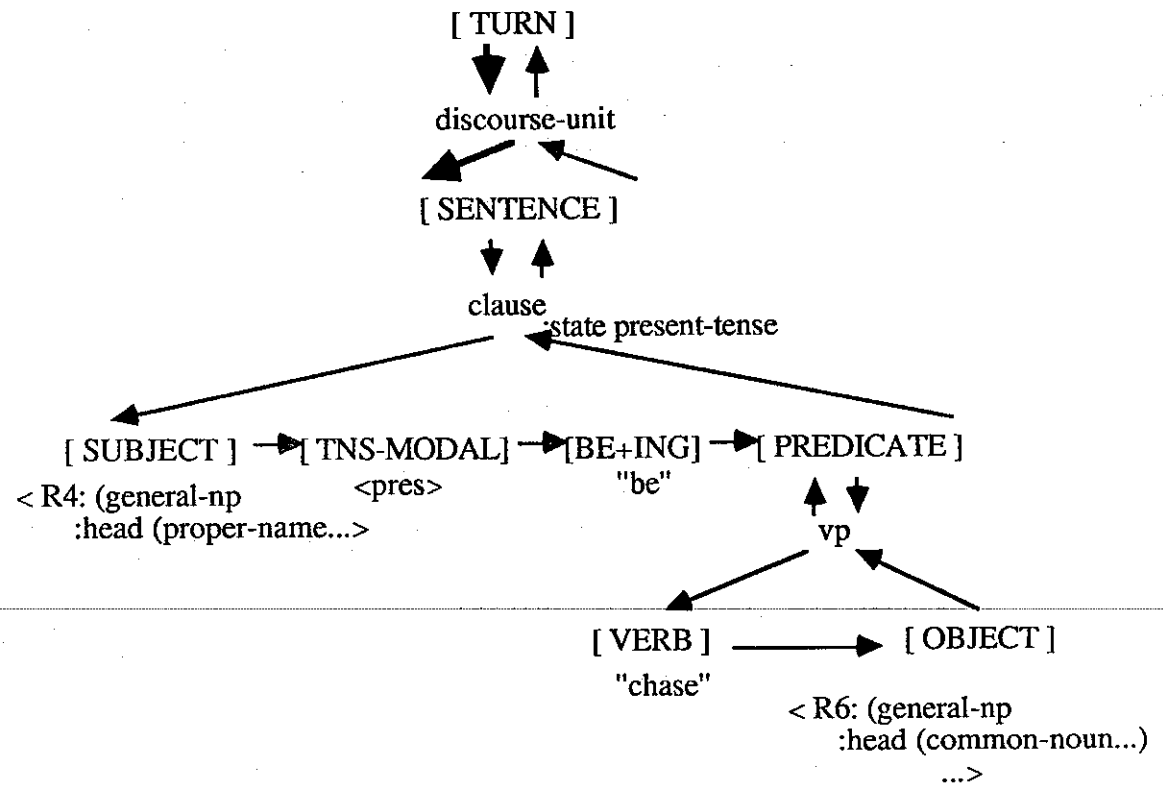


FIGURE 4.14

4.5 Realizing the subject

Phrase structure exection continues its traversal, passing through the clause node to the slot labeled SUBJECT. The situation is now the same as it was when we were at SENTENCE above: the current position is a slot whose contents is a unrealized specification. Control goes to the realization process to realize the specification, repeated for convenience in Figure 4.15.

```
(general-np
  :head (np-proper-name "Fluffy")
  :accessories (:number singular
                :determiner-policy no-determiner))
```

FIGURE 4.15

The bundle type is general np, shown with its driver in Figure 4.16. This driver first checks to see if the context requires obligatory pronominalization, since no further specifications would be realized if that were the case. Obligatory pronominalization rules include c-command (Peter bought *his* friend a book) and reflexives (The barber shaves *himself*). If there is no reason to pronominalize, the bundle is realized in much the same way as the general clause bundle: first the head is realized, then accessories are processed, and finally further specifications are attached in.

```
(define-bundle-type  GENERAL-NP
                     general-np-bundle-driver
  accessory-list (:number
                  :person
                  :gender
                  :determiner-policy ))

(defun GENERAL-NP-BUNDLE-DRIVER (B)
  (let ((reason-to-pronominalize
         (should-be-pronominalized-in-present-context (underlying-object B))))
    (if reason-to-pronominalize
        (then(select-appropriate-pronoun B reason-to-pronominalize))
        (else
          (let ((np-root (realize (head B))))
            (unless (pronounp np-root) ;; i.e. one that's deliberately chosen
            (set-backpointer-of-root  np-root B)
            (process-number-accessory
              np-root (get-accessory-value :number B t))
            (process-determiner-accessory
              B np-root (get-accessory-value ':determiner-policy B t))
            (let-with-dynamic-extent
                ((current-phrasal-root (context-object np-root)))
              (process-further-specifications (further-specifications B)
                ))
            np-root)))))
```

FIGURE 4.16

In this case the head is a kernel specification with realization function NP-PROPER-NAME and argument "Fluffy". Since there are no choices in the linearization of a single argument, the realization function is a single choice, shown in Figure 4.14. Grammatical characteristics are checked against grammatical constraints on the position to insure grammaticality (see subject label, Figure 4.17) and then the phrase (see definition, Figure 4.17) is built and the argument of the kernel is mapped to the np-head position. Processing the accessories involves setting the state of the np in the phrasal context. State for an np is a vector of number/determiner-policy.

```
(define-single-choice NP-PROPER-NAME
  :phrase  proper-name
  :grammatical-characteristics (np) )

(define-slot-label SUBJECT
  grammatical-constraints (np)
  associated-attachment-points (tense-modal))

(define-phrase PROPER-NAME (n)
  (np
    np-head n  :additional-labels (proper-noun) ))
```

FIGURE 4.17

Again, there are no futher specifications, so the next step is to splice the np into the subject position and return control to phrase structure execution.

```
(define-node-label NP
  word-stream-actions ((determiner initial))
  associated-attachment-points (possessive))
```

FIGURE 4.18

Phrase structure execution first enters the node np (see figure 4.18) which has an associated initial word-stream-action for the determiner. In this case the determiner state is NO-DETERMINER, so no action is taken. Next it moves to the np-head position. Since the contents of that position is a word (see Figure 4.19), the word is sent to morphology (no action is taken in this case as it is a proper name) and the word is output.

```
(define-word "Fluffy" (proper-noun) )
```

FIGURE 4.19

Next, PSE returns to the node NP, then to the slot SUBJECT, then on to the TENSE-MODAL position. Figure 4.20 shows the path traversed so far in bold and the current output.

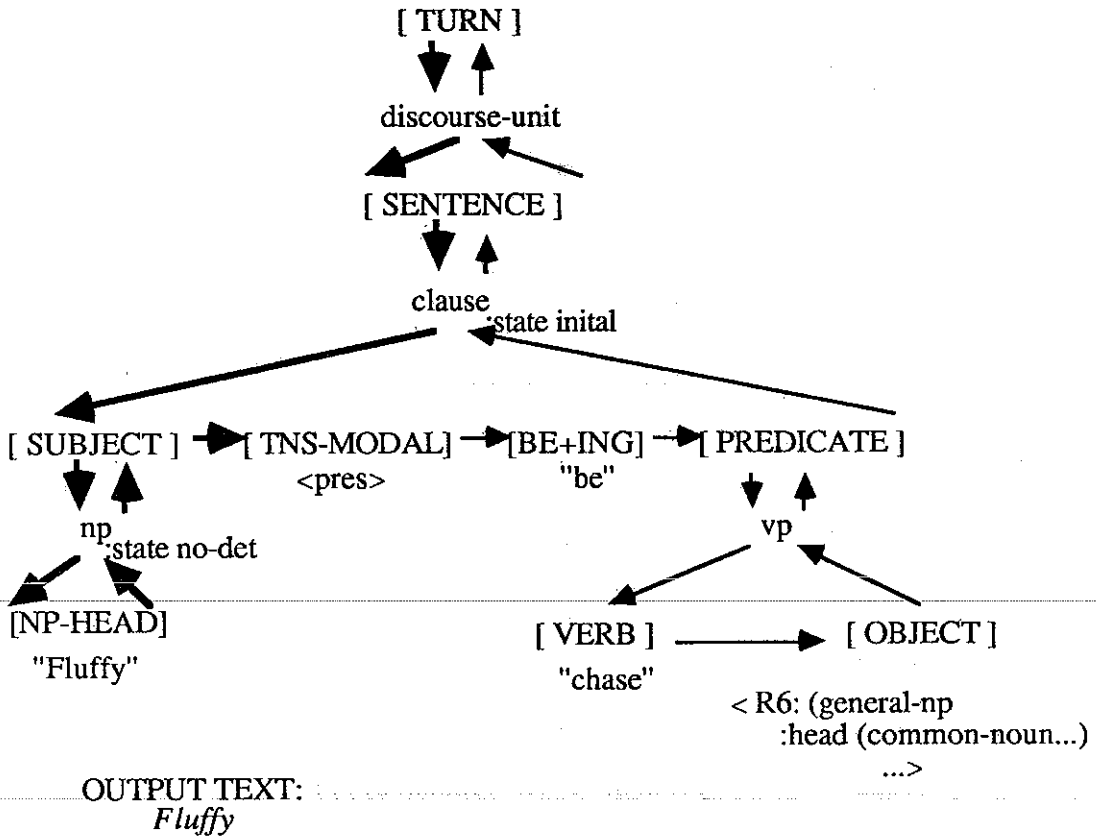


FIGURE 4.20

4.6 Realizing the verb group

As discussed in section 2.7, the morphology of the verb group is handled by a state machine. Actions are controlled by three things: The state of the current phrasal root (the CLAUSE node in this case), the label of the current position, and the contents of the current position. As we enter the verb group in this example, the state is INITIAL, the label on the current position is TENSE-MODAL, and its contents is the tense-marker PRESENT. The function PROCESS-TENSE is called (shown in figure 4.21) which changes the state to PRESENT-TENSE.

```
(defun PROCESS-TENSE (tense-marker)
  (ecase (verb-group-state)
    (initial
      (case (name tense-marker)
        (past (set-verb-group-state 'past-tense))
        (present (set-verb-group-state 'present-tense))))
    (prepose-aux
      (case (name tense-marker)
        (past (set-verb-group-state 'past-tense))
        (present (set-verb-group-state 'present-tense)))
      (induce-auxiliary)
      (set-verb-group-state 'unmarked))))
```

FIGURE 4.21

Next PSE moves to the position labeled BE+ING, with contents "be". VERB-MORPHOLOGY is called (since "be" is a verb) which calls PROCESS-PROGRESSIVE (since the label on the current position is BE+ING). (See Figure 4.22.)

```
(defun VERB-MORPHOLOGY (contents slot-name)
  (ecase slot-name
    (tense-modal (process-modal contents))
    (have+en     (process-perfective contents))
    (be+ing      (process-progressive contents))
    (be+en       (process-passive contents))
    (verb        (process-verb contents))))

(defun PROCESS-PROGRESSIVE (verb)
  (case (verb-group-state)
    (past-tense (past-tense-form verb))
    (present-tense (present-tense-form verb))
    (unmarked (send-to-output-stream (pname verb)))
    (perfective (en-form verb))
    (otherwise (mbreak "error in process-progressive")))
  (set-verb-group-state 'progressive))
```

FIGURE 4.22

Morphology then adds the present tense form of the verb (since the current state is PRESENT-TENSE). In this case, since "be" is irregular (see Figure 4.23), it looks up the appropriate form and outputs "is".

```
(define-word "be" (verb)
  present-1st-singular "am"
  present-2nd-singular "are"
  present-3rd-singular "is"
  present-plural "are"
  past-1st-singular "was"
  past-2nd-singular "were"
  past-3rd-singular "was"
  past-plural "were"
  en-form "been"
  ing-form "being" )
```

FIGURE 4.23

PSE continues its traversal through PREDICATE and VP to the position labeled VERB, with contents "chase". VERB-MORPHOLOGY is again called (since "chase" is a verb); it calls PROCESS-VERB (since the label on the current position is VERB). (See figure 4.24). Morphology then adds the ing-form to the verb (since the current state is PROGRESSIVE) and outputs the word "chasing".

```
(defun PROCESS-VERB (verb)
  (case (verb-group-state)
    (past-tense (past-tense-form verb))
    (present-tense (present-tense-form verb))
    (unmarked (send-to-output-stream (pname verb)))
    (perfective (en-form verb))
    (progressive (ing-form verb))
    (passive (en-form verb))
    (initial (send-to-output-stream "to")
             (send-to-output-stream (pname verb)))
    (otherwise (mbreak "error in process-verb")))
  (set-verb-group-state 'initial))
```

FIGURE 4.24

Figure 4.25 shows the path traversed so far in bold and the current text output:

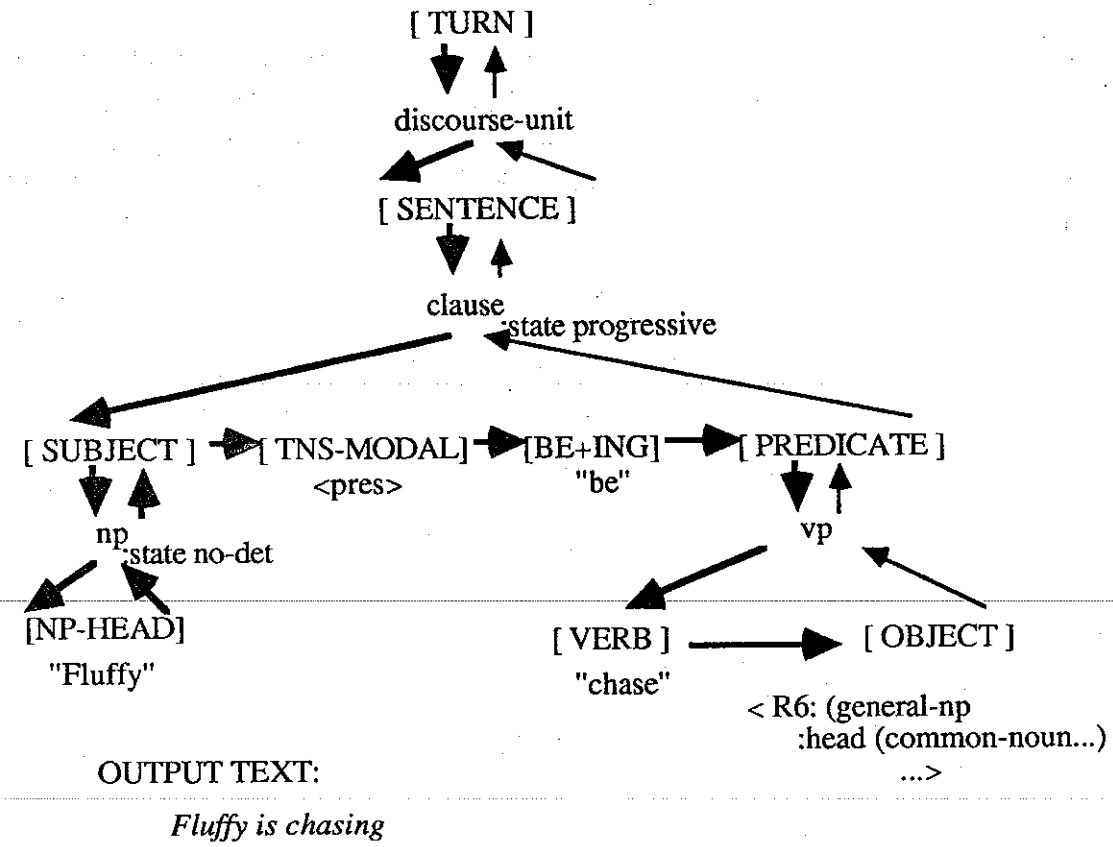


FIGURE 4.25

4.7 Realizing the object

Now the system is once again in the state at being at a position whose contents is an unrealized specification. The realization specification <R6> from Figure 4.1 is repeated below in Figure 4.26 for convenience.

```
(general-np  <R6>
  :head (np-common-noun "mouse")  <R7>
  :accessories (:number singular
                :determiner-policy indefinite-first-mention)
  :further-specifications
    ((:specification
        (predication_to-be *self*  <R8>
                           (adjective "little") <R9>)
      attachment-function restrictive-modifier)))
```

FIGURE 4.26

The head specification is realized and accessories processed in the same way as for the specification <R4> ("Fluffy") described above. The realization function, phrase, and result of its instantiation is shown in Figure 4.24. However, in this case, there

are further specifications that must be processed before the phrase is knit into the tree.

```
(define-single-choice NP-COMMON-NOUN
  :phrase  common-noun
  :grammatical-characteristics (np) )

(define-phrase COMMON-NOUN (n)
  (np
    np-head n  :additional-labels (common-noun) ))
```

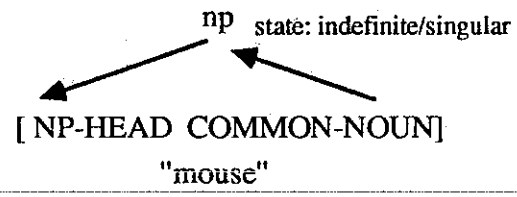


FIGURE 4.27

A further specification in a bundle has two parts (see section 2.2): A specification, which may be a kernel or a bundle specification, and an attachment function. In much the same way as a realization function defines the possible phrasal realizations for a kernel, an attachment function defines the possible points in the surface structure a specification may be attached. In this case, the attachment function is a class: restrictive-modifier (shown in Figure 4.28). An attachment class does not have characteristics to distinguish the choices, as does a realization class, but rather uses the active attachment point list and the constraints on the specification to be attached into differentiate between them.

```
(define-attachment-class RESTRICTIVE-MODIFIER ()
  ((ADJECTIVE)
   (RESTRICTIVE-APPOSITIVE)
   (RESTRICTIVE-RELATIVE-CLAUSE)
   (NP-PREP-COMPLEMENT )))
```

FIGURE 4.28

This class consists of four attachment points, each defined in terms of a reference label, a link (e.g. previous or next), and the label to be put on a new slot spliced into the surface structure (see Figure 4.29).

```
(define-splicing-attachment-point ADJECTIVE
  reference-labels (common-noun)
  link (previous)
  new-slot (adjective))

(define-splicing-attachment-point RESTRICTIVE-RELATIVE-CLAUSE
  reference-labels (common-noun)
  link (next)
  new-slot (relative-clause))
```

FIGURE 4.29

4.8 Choosing an attachment point

The first filter on the class uses the active attachment point list. When the initial phrase was instantiated, the attachment points associated with the labels on the phrase were gathered up into a list, ACTIVE-ATTACHMENT-POINTS, associated with the phrasal root. In this case, all four of the attachment points in the class are active, as they all appear in the labels which make up the phrase (shown in Figure 4.30).

```
(define-slot-label NP-HEAD
  associated-attachment-points
    (non-restrictive-relative-clause non-restrictive-appositive
     np-prep-complement))

(define-slot-label COMMON-NOUN
  associated-attachment-points
    (adjective restrictive-relative-clause restrictive-appositive))

(define-node-label NP
  word-stream-actions ((determiner initial))
  associated-attachment-points (possessive))
```

FIGURE 4.30

The next filter on the class compares the available choices defined by the realization function of the specification with the constraints imposed by the linguistic context if that attachment point were used. In this case the realization function is the realization class PREDICATION_TO-BE (shown in figure 4.31). The grammatical characteristics of each choice are compared with the grammatical characteristics on the new slot used by each attachment point and the argument characteristics check the relation of an argument with elements already realized.

```
(define-realization-class PREDICATION_TO-BE (o p)

   1  ((S-be-COMP o p)
       :grammatical-characteristics (clause)
       :required-accessories (:unmarked))

   2  ((p)
       :argument-characteristics (identical-with-root o))

   3  ((S-be-COMP-subj-rel o (o :trace) p)
       :grammatical-characteristics (relative-clause)
       :argument-characteristics (identical-with-root o))

   4  ((S-be-COMP-comp-rel p o (p :trace))
       :grammatical-characteristics (relative-clause)
       :argument-characteristics (identical-with-root p))

   . . . )
```

FIGURE 4.31

Let's look at each of the choices in turn:

Choice 1 is *not* acceptable, since none of the attachment points in the class builds a slot with grammatical constraints CLAUSE.

Choice 2 is acceptable. Since it is a parameter, the check is a bit more complicated: the grammatical constraints on the *value* of the parameter must be checked; in this case the value of the parameter **p** is the specification (ADJECTIVE "LITTLE"); There is an acceptable match since the grammatical characteristics on the single choice ADJECTIVE and the grammatical constraints on the slot label ADJECTIVE are both PRENOMINAL-MODIFIER. (See Figure 4.32.) Argument characteristics check that the source of the root of the phrase (which is the specification that built the phrase <R7>) is identical with the value of the argument bound to the parameter **O** (i.e. *self* in the specification in Figure 4.26).

Choice 3 is acceptable, since RESTRICTIVE-RELATIVE-CLAUSE has the grammatical constraints relative clause and again the argument bound to **o** is identical with the root of the phrase.

Choice 4 is *not* acceptable, since although RESTRICTIVE-RELATIVE-CLAUSE has the grammatical constraints relative clause, the argument bound to **p** is *not* identical with the root of the phrase.

```
(define-slot-label ADJECTIVE
  grammatical-constraints (prenominal-modifier))

(define-single-choice ADJECTIVE
  :choice-is-argument
  :grammatical-characteristics (prenominal-modifier))
```

FIGURE 4.32

This makes both the ADJECTIVE and RESTRICTIVE-RELATIVE-CLAUSE valid attachment points for attaching this specification. ADJECTIVE is chosen because it is first in the class.[2] The slot labeled adjective is spliced into the tree before the np-head. Next the specification is "specialized", that is the realization function is narrowed to include only those choices found acceptable by the filtering process just run. In this case, specialization takes the class down to a single choice. Since in this case the choice is a parameter (rather than a phrase as in the previous examples), specialization includes making the choice the value of that parameter, in this case the specification (adjective "little"). (Note: the value of a parameter will always be either a specification or a word. The specification is now placed as the contents of the new slot. The result is shown in Figure 4.33.

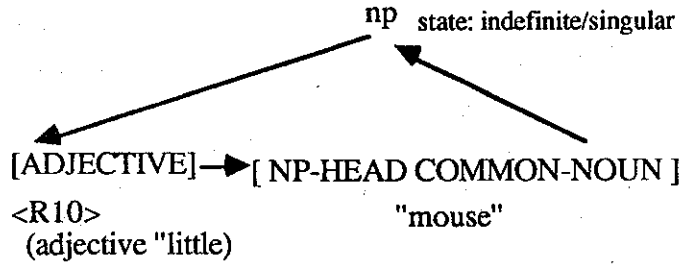


FIGURE 4.33

4.9 Finishing

This result is knit into the surface structure tree and phrase structure exectution takes over. At the NP node, PSE executes the word stream action DETERMINER which call the function PRINT-DETERMINER (Figure 4.34). Since the : DETERMINER value of the state of the np is INDEFINITE and the :NUMBER value of the state is SINGULAR, "a" is output.

---

[2] We would like to add stylistic preferences and other more subtle distinctions, rather than relying on merely order in these cases. Order relies too heavily on the grammarian building the classes, rather than positive criterion.

```
(defun print-determiner (state-list)
  (ecase (state-value :determiner statelist)
    (definite (send-to-output-stream "the"))
    (indefinite (ecase (state-value :number state-list )
                  (singular (send-to-output-stream "a"))
                  (plural))
   (no-determiner)))
```

FIGURE 4.34

The next position is the slot ADJECTIVE, whose contents is a specification. The realization function is the single-choice ADJECTIVE (shown in Figure 4.32), whose choice is simply the argument, in this case the word "little". Now we are still at the slot, but the contents is a word, so it is sent to morphology and (as there are no inflections on the adjective) output. The next position is the np head, whose contents is the word "mouse". It is sent to morphology and since the state of the NP is SINGULAR, it is output without inflection. Note that had the state been PLURAL, morphology would have looked up the irregular form of the plural associated with the word (see figure 4.35) and output "mice".

```
(define-word "mouse" (noun) plural "mice" )
```

FIGURE 4.35

PSE continues its traversal back up the tree, taking no actions until it reaches the SENTENCE slot (whose label is repeated in Figure 4.36), which has a FINAL word stream action.

```
(define-slot-label SENTENCE
  grammatical-constraints (clause)
  word-stream-actions ((punctuation final period)
                       (capitalize-the-next-word initial)))
```

FIGURE 4.36

It then places the period at the end of the sentence. The result is shown in Figure 34.

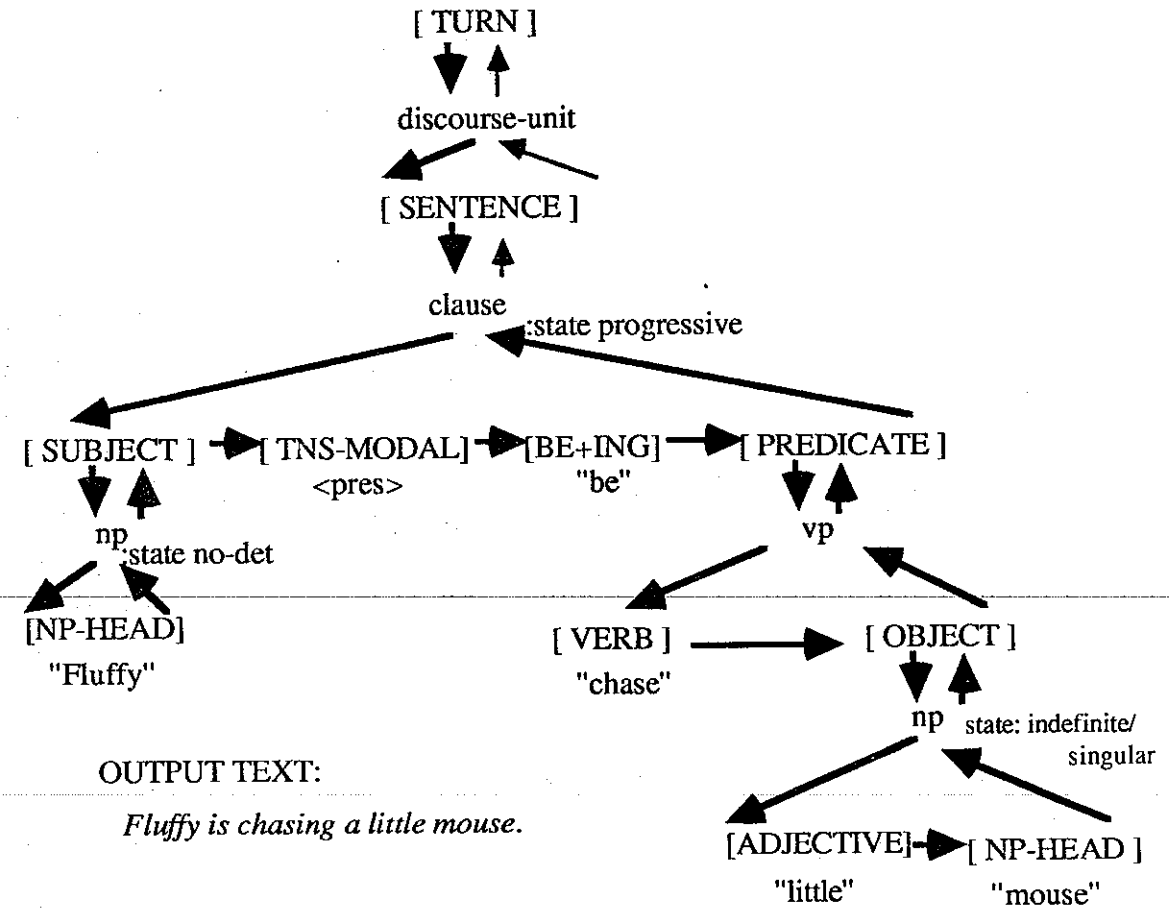


FIGURE 4.37

When PSE reaches the node DISCOURSE-UNIT, it checks to see if there are specifications on the PENDING-RSPECS list (in this case there are none--if there had been, it would have attached the first in the list as NEXT-SENTENCE). When it reaches the slot TURN the process terminates.

5. MANAGING MUMBLE

Contents

5.1 DIRECTORIES

This section describes Mumble's directory tree. Under each subdirectory we give a short overview of the directory's contents and a description of the contents of any files in the directory whose names are not self explanatory. We then list the file names.

Note: in many directories, there is a file called DESIGN.LISP. This file has two functions: to define the appropriate package for this directory, and to give an overview of the contents of the directory. The design files may be useful to people walking through the Mumble tree.

GRUMPY:>mumble-86>*.*

The top level directory contains the files for starting up Mumble. LISPM-INIT allows the user to log in as Mumble-86 (on systems where Mumble-86 can be defined as a user) and begins the loading process automatically. The BOOTSTRAP file loads the Loader (see section 5.2) and all of the package definitions. VERSION-DB contains the definition of what files constitute the "standard Mumble" version and their loading

sequence. CHANGELOG is a record of all file by file modifications to Mumble's code that have been made since the last major version of the system was established.

top-level files:
bootstrap.lisp
changelog.text
design.lisp
lispm-init.lisp
version-db.lisp

GRUMPY:>mumble-86>browser>*.*

The browser directory contains all the files which are involved in using the windowing facilities of a Lispmachine (Symbolics or TI) to show Mumble's activities and data structures. CONFIGURATIONS defines the constraint frame, i.e. the alternative configurations of panes. MUMBLE-FONT.KST defines a special font for a menu for selecting configurations; presently this is only functional on TI Explorers. APPLICATION-WINDOW-FLAVORS, COMMAND-MENU, and PROCESS-PANE define the different panes that make up the browser. The designer is discussed in section 3.2.3.

application-window-flavors.lisp
command-menu.lisp
configurations.lisp
constraint-frame.lisp
interface-to-text-output.lisp
message-pretty-printer.lisp
mumble-font.kst
process-pane.lisp
rclass-pretty-printer.lisp
surface-structure-pretty-printer.lisp

GRUMPY:>mumble-86>browser>line-buffer-geometry>*.*

"Line buffer geometry" is a largely device independent facility for supporting customized pretty printing of mouse sensitive objects. It is discused as part of the Browser, section 3.2.3.

blinker.lisp
display-engine.lisp
insert-new-objects.lisp
lines.lisp
mouse-click.lisp
mouse-moves.lisp
virtual-display-items.lisp
window-flavor.lisp

GRUMPY:>mumble-86>grammar>*.*

This directory contains the definitions of the objects that make up Mumble's grammar. Files are named for the type of the objects defined in them. For a

discussion of the object types and the relationship between the definitions and the internal Mumble objects, see section 3.1.1. For a discussion of the entire type system, see Section 3.2.1.

attachment-classes.lisp
attachment-points.lisp
characteristics.lisp
labels.lisp
morphology.lisp
phrases.lisp
pronouns.lisp
punctuation-marks.lisp
tense-markers.lisp
traces.lisp
word-stream-actions.lisp
words.lisp

GRUMPY:>mumble-86>interactive-demo>*.*

The interactive demo is a facility which allows the user to modify and run input specifications to Mumble. It is described in detail in Section 7.

command-menu.lisp
edit.lisp
globals.lisp
help.lisp
options.lisp
pretty-printer.lisp
save.lisp
utility.lisp

GRUMPY:>mumble-86>interface>*.*

The interface code builds input specifications to Mumble both from underlying applications programs and from "stand alone" demos.

GRUMPY:>mumble-86>interface>applications>*.*

At top level in this directory is the code for created the table linking model level objects and templates which build Mumble input specifications, and the code for inststantiating that mapping. The code particular to an application is kept in a subdirectory named for it. Three current projects are included here: Genaro, MT (Mumble-TEXT), and Apt. These projects are all in progress, so this code is in flux.

For a more detailed description of the issues in building an interface to Mumble, see Section 6.

```
instantiating-specifications.lisp
object-to-specification-table.lisp
>apt>*.*
    apt-noun-phrases
    first-sentence-test
    idiosyncratic-templates
    location-templates
    there-is-object
    x-hand-side
>genaro>*.*
    design.lisp
    idiosyncratic-templates.lisp
    interface-utilities.lisp
    model.lisp
    test-cases.lisp
    trivial-default-specifications.lisp
>mt>*.*
    mappings.lisp
    templates.lisp
```

GRUMPY:>mumble-86>interface>stand-alone>*.*

Our "stand alone" interface allows the user to create "demo" input specifications. It is used for testing grammatical constructions or to run demonstrations of Mumble without any planner or application program behind it. SPECIFICATION-LANGUAGE contains the function CREATE-MESSAGE, which turns the symbols and lists of a stand alone specification into the actual Mumble objects (bundles, kernels, accessories, etc) which make up the input specification language.

```
defining-demos.lisp
interface-utilities.lisp
rspec-table.lisp
specification-language.lisp
>demos>*.*
    demos-to-use.lisp
    fluffy-demos.lisp
    fluffy-via-templates.lisp
```

GRUMPY:>mumble-86>interpreters>*.*

The interpreters directory contains the code for the main processes of the "virtual machine": Realization, Attachment, and Phrase Structure Execution. (See sections

2.3 - 2.6 and 3.2.2) It also contains the function MUMBLE (in TOP-LEVEL) which begins the program's execution.

attachment.lisp
phrase-structure-execution.lisp
position-path-operations.lisp
state.lisp
text-output.lisp
top-level.lisp
GRUMPY:>mumble-86>interpreters>realization>*.*
    instantiate-phrase.lisp
    realize.lisp

GRUMPY:>mumble-86>lisp>*.*

In this directory contains Lisp utility code, for example code for compatibility between Symbolics and TI Lispmachines, and Common-lisp implementations of useful ZetaLisp functions.

compatibility.lisp
configuration-specs.lisp
design.lisp
glass-tty-menus.lisp
independent-command-menu.lisp
menu-utilites.lisp
misc.lisp
object-db.lisp
patch-to-load-function.lisp
pathnames.lisp
sorry.lisp
system-specific.lisp
world-load.lisp

GRUMPY:>mumble-86>loader>*.*

In this directory is code for loading Mumble. SPECIFYING-VERSIONS contains code to allow different versions of Mumble to be defined. (A "version" in this sense is a specification one set of files rather than another, all of them existing at the same time, e.g. as might be used by different people experimenting with variations on the standard system). For more information on the loader, see Section 5.2

design.lisp
loader.lisp
preloader.lisp
specifying-versions.lisp

GRUMPY:>mumble-86>message-level>*.*

This directory contains code for both "sides" of the message level: the definition of the objects used in a specification and the code for building specifications from predefined templates. Objects used in the input specification language include realization functions (REALIZATION-CLASSES, CURRIED-REALIZATION-CLASSES,

and SINGLE-CHOICES), and BUNDLE and ACCESSORY TYPES. (See sections 2.2) SPECIFICATION-TEMPLATES used in building specifications use code in CONSTRUCTING-BUNDLES and OPERATORS-OVER-SPECIFICATIONS. (See section 6.)

accessory-processing.lisp
accessory-types.lisp
bundle-drivers.lisp
bundle-types.lisp
constructing-bundles.lisp
curried-realization-classes.lisp
operators-over-specifications.lisp
realization-classes.lisp
single-choices.lisp
specification-templates.lisp

GRUMPY:>mumble-86>objects>*.*

The special forms for defining all the object types used in Mumble and their postprocessing functions are included in this directory For a more complete discussion of the type system and the various object types, see sections 3.1.1 and 3.2.1.

all-the-object-types.lisp
postprocess-objects.lisp
postprocessing-order.lisp
short-printers.lisp

GRUMPY:>mumble-86>tracker>*.*

The tracker displays ongoing information about Mumble's processing. It is triggered by landmarks which are placed throughout the Mumble code. For discussion , see Section 3.2.2.

datatypes.lisp
define-landmark.lisp
define-trackable-datatype.lisp
design.lisp
execution.lisp
global.lisp
initialization.lisp
program-interface.lisp
sets.lisp
specials.lisp
user-interface.lisp

GRUMPY:>mumble-86>types>*.*

This directory contains the code for the special forms used to define objects. It is discussed in section 3.2.1.

creating-objects.lisp
defining-types.lisp
design.lisp
postprocessing.lisp

5.2 THE LOADER

Mumble has its own "virtual file system" and "version definitions," which are similar to the "logical pathnames" and "DEFSYSTEMs" available on Lisp Machines, but with slightly different functionality. They accomplish the following:

1) A divorce between the physical directory structure at the site and the "virtual directories" used in defining versions of Mumble, which means you don't have to organize your directories the same way we do at UMass. This is essentially what logical pathnames do; the difference is that a virtual directory can map to more than one physical directory, which is not the case with logical pathnames.

2) Multiple Versions of Mumble. Often, someone will want to have a personal version of Mumble in which has some experimental changes. Copying all of Mumble to another place just to change a few files is unnecessarily wasteful. What we would like to do is say "Load Mumble's virtual files, but for each file check in my directories before checking the standard directories and prefer the files in my directories." Thus, the personal files "shadow" the standard files. The shadowing is done by mapping a virtual directory to more than one physical directory and taking the file out of the first physical directory to supply one. This is very similar to the Unix(tm) "path" mechanism.

3) The ability to re-load Mumble without re-booting and starting from scratch, or to save Mumble in a "world" (band) and, after booting into it, load only those files that are newer than the band. To do this, the loader simply keeps a record of what files have been loaded, so that if the LOAD-MUMBLE function is called again, it can skip the loading of files that have already been loaded.

5.2.1 Defining Versions

A version of Mumble consists mainly of a set of file groups (which are something like the "modules" in a DEFSYSTEM) and a sequence of "directory

indices." A "directory index" is a mapping from virtual directories to physical directories, and you can use as many of these in defining your version as you wish. For instance, you can have the loader search first your own directories, second your research group's directory of experimental files, and lastly the standard Mumble files (as distributed from UMass, say). Such a Mumble version definition might look like this:

```
(define-mumble-version user::my version
                       (my-directories standard-mumble-directories)
  (:type-system       type-system)
  (:objects     object-type-definitions)
  (:tracker     tracker)
  (:grammar     grammar message-level message-level-part-two)
  (:interface   stand-alone applications demos ALBM)
  (:interpreters      interpreters)
  (:browser     line-buffer-files browser interactive-demo))
```

The first argument is the name, which is a symbol that's used in constructing menus and such. Second is the list of directory indices; these are just the names, the indices must be previously defined. The rest is a list of clauses which specify parts of the Mumble code and what file groups compose them. The loader loads each of these clauses in a fixed order; for instance, the type system is loaded before the object definitions, which are loaded before the grammar. There is no way to denote load-order dependencies between files or file-groups---we've just built this knowledge into the loader.

5.2.2 Defining Directory Indices

A directory index is a mapping from a symbol, the name of the virtual directory, and a physical directory. This mapping is internally implemented as an association list, and the syntax of the defining form is similar to an association list. For example:

```
(define-directory-index  standard-mumble-directories
  :root "mumble-86:"
  (grammar          "grammar")
  (message-level    "message-level")
  (demos            "interface" "stand-alone" "demos")
  (test-demos       "tests" "demos")
  (test-grammar     "tests" "grammar")
  (stand-alone      "interface"  "stand-alone")
  (interactive-demo "interactive-demo")
  (applications     "interface"  "applications")
  (interpreters     "interpreters")
  (interp-realiz    "interpreters" "realization")
  (objects          "objects")
  (ALBM             "interface" "applications" "ALBM")
  (genaro           "interface" "applications" "genaro")
  (mt               "interface" "applications" "mt")
  (browser          "browser")
  (line-buffers     "browser" "line-buffer-geometry")
  (tracker          "tracker")
  (types            "types"))
```

One feature is the "ROOT" pathname. All the directories are treated as subdirectories of the root. The root is also how you specify the host and device for the virtual directories. The rest of the form is just a sequence of lists, where the car is a symbol, the name of a virtual directory, and the cdr is a list of strings, the components of the directory name. Whether you're on a VMS machine and the virtual directory A maps to [foo.bar.baz], or on a Symbolics and A maps to >foo>bar>baz>, list will be (A "foo" "bar" "baz"). Warning: Don't forget the closing punctuation on the root pathname, in this case a " > "; it's easy to do that and the resulting error message is unhelpful.

5.2.3 Shadowing Directory Indices

Very often when you have your own version, you will have your files in your own directory, but organized the same way Mumble's are (less taxing to remember). To make it very easy to do this, there is a handy macro:

```
(define-shadowing-directory-index
  sda-directories
  standard-mumble-directories
  "vax1:nlg$disk:[anderson.my-mumble]")
```

This defines a directory index named SDA-DIRECTORIES which are to be exactly like the STANDARD-MUMBLE-DIRECTORIES, except that the root is a sub-directory of my Vax account.

Typically, this personal version definition is put is a file by itself, together with personal definitions of file-groups and so forth. Then, the global variable USER::*PERSONAL-VERSION-DEFINITION-FILE* is set to the name of that file, that

is, it is set to a string acceptable to the LOAD function. Mumble's loader checks this variable before loading Mumble, to see if it is bound, and if it is, it loads the specified file.

5.2.4 Defining File Groups

A File Group is very simple; it's just a sequence of virtual files and other file groups (allowing nesting is occasionally useful and was easy to implement). Here's an example:

```
(define-file-group BROWSER ()
  (configurations                    browser)
  (constraint-frame                  browser)
  (command-menu                      browser)
  (process-pane                      browser)
  line-buffer-files
  (application-window-flavors        browser)
  (message-pretty-printer            browser)
  (rclass-pretty-printer             browser)
  (surface-structure-pretty-printer browser))
```

Just to keep you alert, we changed the usual order of filename components---the name of the file comes first and the name of the virtual directory comes second. Thus, BROWSER had better appear in at least one of the directory indices we defined, and CONFIGURATIONS, CONSTRAINT-FRAME, and so forth are files in that directory. (Actually, the non-standard syntax is an experiment based on the principle that interesting information should come first.) Note the embedded file group, LINE-BUFFER-FILES. The second argument should be left nil; it's for annotations on the file-group and is currently unused.

5.2.5 Loading Mumble

There is a special "bootstrap" file which gets the whole ball rolling. That file has minimal functionality, which means it doesn't try to do anything smart about seeing whether a file has been loaded or recording that it has; it just plows on, loading files. It defines the MUMBLE-86 logical pathname, loads files to set up the various packages, loads the files of utility functions into the MG-LISP package, and loads the loader in the MG-LOADER package. Then, it gets the loader going by calling the function LOADER, which calls LOAD-MUMBLE-FROM-SCRATCH, which calls lOAD-MUMBLE. It's the last two, functions which are interesting.

LOAD-MUMBLE-FROM-SCRATCH binds to T a dynamic variable LOADING-WHOLE-SYSTEM which is accessed by the type system. If that variable is T, the type

system delays all postprocessing of objects until after all the grammar files are loaded and then sweeps through all type lattice doing all the postprocessing.

LOAD-MUMBLE is the real workhorse. First, if its optional argument is T (LOAD-MUMBLE-FROM-SCRATCH calls LOAD-MUMBLE this way), it loads the version definition files. One of these is the standard "version-db" file which defines all the standard Mumble file groups, directory indices, and versions. The other, which is optional, is the value of USER::*PERSONAL-MUMBLE-VERSION-DEFINITION-FILE*, which was described above. If that symbol is bound, if should be bound to the name of a file. The Loader will then read version definitions from that file. (Actually, it just "loads" the file, so you could put any arbitrary Lisp code in there, though there's little point.) This symbol is in the user package because you would ordinarily set it before loading the bootstrap file, at which time none of the Mumble packages will exist.

Second, LOAD-MUMBLE figures out which version of Mumble you want load. Ordinarily, it just throws up a menu of all the possible versions and you click on one. However, this can be short-circuited by setting the variable USER::*VERSION-OF-MUMBLE-TO-LOAD* to the name of the version to be loaded. It's important to remember that the Loader just looks through its list of versions looking for one whose name is EQ to the value of this variable, so *the names have to be in the same package*. The USER package is the easiest one to put the names in, since it always exists. But the version-definition file will usually be in the MUMBLE package, so that all the defining forms are available. The solution is to simply define the version with a qualified name, as shown in the example above.

LOAD-MUMBLE does all the loading of file groups next, but that need not concern us.

Loading a File Group Separately

Normally, everything is loaded directly by LOAD-MUMBLE, but you may occasionally want to load some file group by hand. The function LOAD-A-FILE-GROUP is the way to go. With no arguments, it throws up a menu of all the file groups and loads the one you click on. Optional arguments let you to tell it to re-load the version definitions (in case you just modified that file to define file group you want to load) and let you tell it whether to load the file group with respect to this version or to query you about the version as well. The version is important because it defines the sequence of directory indicies that will be searched.

5.2.6 Reloading Mumble

If Mumble is already loaded, but the software that is loaded is slightly out of date because new files have been written, you can re-load Mumble. Simply call either LOAD-MUMBLE-FROM-SCRATCH or LOAD-MUMBLE. As the Loader goes through the files to be loaded, it checks against a list of loaded files and only loads a file if it has not previously been loaded. This functionality is similar to loading patches to a DEFSYSTEM, except that nothing need be marked as a patch.

Wanting to re-load Mumble can come about in two ways. You might have done a great deal of editing and simply decide to save all your buffers and re-load. While this is probably slower than evaluating all your buffers, it does afford the chance to test that the version is defined properly (that is, your version definition file is correct) without having to re-boot and start from nothing. The more frequent reason to re-load Mumble is when you have booted into a saved world (a "band") that included Mumble, but Mumble from before certain files were changed. Rather than remember or guess which files need to be loaded after booting, you can make the Loader run through all the files. If none need to be loaded, the whole process takes a little over a minute and a half, so there is still a great advantage to making a band and running LOAD-MUMBLE (or LOAD-MUMBLE-FROM-SCRATCH) after booting into it.

Warning: if there have been changes to any of the files loaded by the bootstrap file (instead of by the Loader), the Loader will not be aware of this and you will not get the newer versions. Possibly, you can remember to load them yourself, probably before calling load-mumble. It may be better to make a new band.

Subtlety Warning: if there have been any changes to all-the-object-types, then that file will be reloaded, causing new type-objects and new catalogs to be created. This means old objects will no longer be in the catalogs. If any files that create those objects are loaded, then the objects will get new definitions and therefore be in need of postprocessing. If you called LOAD-MUMBLE-FROM-SCRATCH, it will wait until everthing is loaded and then postprocess *the objects that are in the catalogs*. But the objects that are in need of postprocessing won't be in the catalogs and therefore won't get postprocessing. This will undoubtedly cause the virtual machine to break because it's got a symbol instead of a real Mumble object, and you'll wonder why. You'll be safe if you don't try to re-load if all-the-object-types has changed---just boot into a clean band and start again .

In general, re-loading doesn't keep track of dependencies. For instance, if you make a big change to the type system, you'll probably need to re-load lots of subsequent files even if they haven't changed themselves. The Loader doesn't notice these effects, so you have to consider them.

5.3 SCREEN CONFIGURATIONS

This section describes the window configurations used by the browser[1]. The browser is used for showing various kinds of information Mumble uses in producing text, for giving demos, for debugging, and for editing. It is described in detail in Section 3.2.3. The browser can be used with different screen configurations. A screen configuration includes a set of panes and possibly a Zmacs editor. The following special-purpose panes are used in the Mumble configurations:

Message Window,
Surface Structure Trace,
Text Output Window,
Process Pane,
Realization Class Window,
Command Menu, and
Lisp Listener.

Section 5.3.1 describes the special-purpose panes in more detail and Section 5.3.2 describes each of the Mumble configurations.

5.3.1 Configuration Panes

In this section, we look at the various panes used in the Mumble screen configurations and describe each of operations available in those panes. When interacting with the panes, remember that useful information is often found in the documentation line at the bottom of the screen.

To **select objects** in the panes Message Window, Surface Structure Trace, and the Realization Class Window, it is possible to move the mouse over the text in the windows and outline the representations of Mumble objects. When an object is outlined, the documentation line at the bottom of the screen will give a short description of the object which is currently being outlined. For example, in Figure 5.1, the curried-realization-class CHASE is outlined in the Message Window and the documentation line displays the printed form, #<CURRIED-REALIZATION-CLASS CHASE>.

[1] Note that the browser only works on a Symbolics 3600 or TI Explorer.

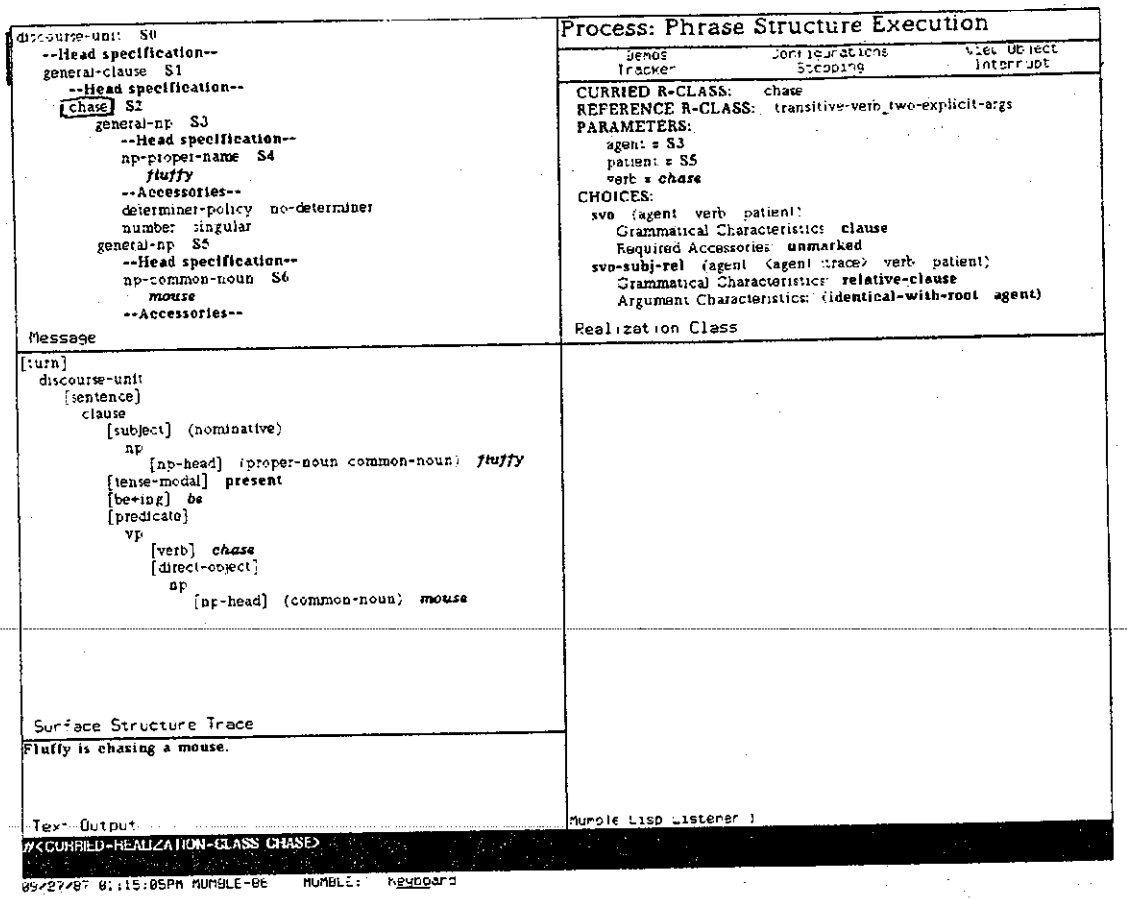


FIGURE 5.1

It is possible to **scroll** the text in the Message Window, the Surface Structure Trace, the Realization Class Window, and the Lisp Listener. To scroll text in any of these panes, slowly move the mouse to the left-hand side of the pane until an up-down arrow appears. The documentation line will then give information about how to scroll the text depending upon which mouse button you click.

The Configuration Panes

The **Message Window** displays the current input specification (the Realization Specification) given to Mumble. The Realization Specification is displayed at the beginning of Mumble's execution and remains static on the screen.

The **Surface Structure** display traces Mumble's construction of its intermediate representation, the surface structure. At the beginning of Mumble's execution, the display in the Surface Structure window is the input message, so the display begins looking like the display in the Message Window. As Mumble executes, the Surface Structure display changes to show the new structures that were selected and instantiated to realize the specifications in the input message.

The **Text Output Window** displays Mumble's output, word by word, as Mumble produces it.

The **Process Pane** indicates the current process which Mumble is executing, either PHRASE STRUCTURE EXECUTION, ATTACHMENT, or REALIZATION.

The **Realization Class Window** allows you to see a description of the Realization Class Mumble is currently using in its processing. During Mumble's execution, when the realization function of a kernel specification is a realization class, that realization class is displayed in the Realization Class window. A realization class consists of a list of parameters and a list of realization class choices. When a kernel specification is being realized, one of the realization class choices is selected as the realization of that kernel specification and the arguments of the kernel specification are mapped to the parameters of the realization class.

The display in the Realization Class Window includes the name of the realization class, the realization class parameters, the kernel specification's arguments which have been mapped to the realization class parameters, and all the choices for the realization class.

The Lisp Listener is useful in many ways when using Mumble. Its most useful function, obviously, is the execution of lisp commands. There is a facility for invoking a Mumble demo as a simple function call in the Lisp Listener. For example, typing: "(fluffy)" invokes the demo named *fluffy*. If you repeat a demo, rather than using the DEMOS command in the Command Menu, you may find it more convenient to use the Lisp Listener's Control-Meta-Y/Meta-Y option which remembers the last command(s) typed to the Lisp Listener. See the Symbolics or TI Explorer manuals for more information on these keystrokes.

The lisp listener also provides a continuous stream of descriptions which are referred to as "landmarks". The landmarks tell you where Mumble is and what Mumble is currently doing. For example, the landmark "Considering choice: #<CHOICE INDEFINITE-NP>" tells you that Mumble is considering an indefinite noun phrase as the realization class choice. Examples of other landmarks appear in the menu in Figure 5.2.

You can turn landmarks on and off by using the TRACKER command in the Command Menu, described in above. The stream of landmarks is also written into a Zmacs buffer named TRACKER OUTPUT, so it can be reviewed, edited, and saved into a file.

The Command Menu

The Command Menu contains the following commands: DEMOS, CONFIGURATIONS, VIEW OBJECT, TRACKER, STEPPING, INTERRUPT, and INTERACTIVE DEMO. These commands allow you to run demos, change

configurations, view Mumble objects, trace Mumble's execution, single-step through its execution, interrupt its processing, and run the Interactive Demo, respectively.

Selecting the **Demos** command produces a menu with a list of demos (input messages) which you can run. After selecting a demo, that demo will run. (Alternatively, you could invoke a demo as a function in the Lisp Listener, as described above.)

Selecting the **Configurations** command brings up a menu showing the different screen configurations available. Selecting one of the configurations will move you to the indicated configuration. For a description of the different configurations, see Section 5.3.2.

The **View Object** command allows you to see any Mumble object printed in the Lisp Listener. Selecting the View Object command displays a menu of all the Mumble object types. Selecting one of the object types brings up a menu of all the objects of that type which are currently defined. Selecting one of those objects will print that object in the Lisp Listener using the "describe" routine. (If no objects of a particular type have been defined, a message will be displayed telling you so.)

Clicking on **Tracker** gives you the following options:

Turn Tracking System ON
Turn Tracking System OFF
Clear current Tracking activity
Set current Tracking activity

The first three options are self explanatory. The fourth option, however, needs a bit more explanation. Selecting the SET CURRENT TRACKING ACTIVITY option brings up the menu in Figure 5.2.

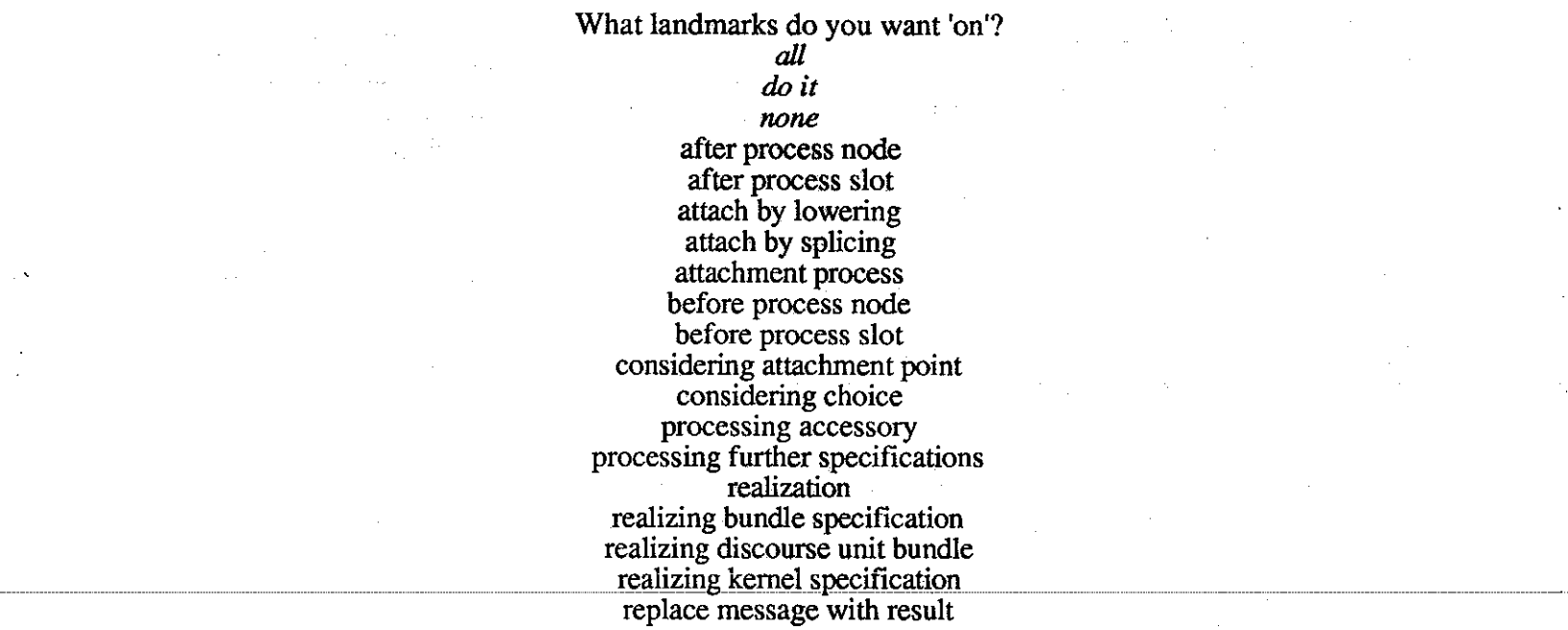


FIGURE 5.2 SET CURRENT TRACKING ACTIVITY MENU

The first three options in the menu, *all*, *do it*, and *none*, work as follows:

all - Clicking on this option turns on all the landmarks listed in the menu.

do it - Clicking on this option turns on only those landmarks which are highlighted in the menu.

none - Clicking on this option turns off all the landmarks in the menu.

The landmarks which are currently 'on' are highlighted in the menu. Clicking on any of the landmarks listed in the menu will toggle its on/off state.

Clicking on **Stepping** allows you to step through Mumble's execution. Many stopping points (landmarks) have been defined in Mumble, and it is these stopping points which you can step through. Different stepping options are allowed depending upon which mouse button you click while on Stepping. Clicking on the left mouse button continues Mumble's execution after a pause; clicking on the middle mouse button turns stepping on; and clicking on the right mouse button turns stepping off. The documentation line displays these options when the mouse is over the Stepping option. The documentation line also displays the current state of stepping, i.e. on or off. The default stepping state is off.

To begin single-stepping through Mumble's execution, click middle on Stepping to turn stepping on. Mumble is currently executing, it will pause after it reaches the next stopping point. To step through Mumble, click left on Stepping (to continue from the pause). Mumble will continue execution until it reaches the next stopping point. To turn stepping off, click right on Stepping.

Clicking on **Interrupt** allows you to pause Mumble at the first stopping point it reaches and subsequently resume from the pause. Clicking the left mouse button on Interrupt pauses Mumble and clicking the middle mouse button on Interrupt resumes from the pause. Again, the documentation line will tell you these options.

The Interrupt option is useful because a lot is happening on the demo screen at once and it is hard to watch all the windows as their displays are constantly changing. Sometimes it is desirable to have a static screen display at various points in Mumble's execution. The Interrupt option allows you to have this static display. Furthermore, the Interrupt option allows you to pause Mumble so you can scroll the displays in the Message Window, the Surface Structure Trace, and the Realization Class Window.

Clicking on **Interactive Demo** will start the Interactive Demo which allows you to interactively edit an input message given to Mumble and to see the effects those changes have on Mumble's output. Section 7 describes the Interactive Demo in more detail.

5.3.2 Screen Configurations

The screen configurations currently available include the follwing:

Demo Configuration,
Zmacs--Lisp--Output Configuration,
Message--Zmacs--Lisp--Output Configuration,
Surface Structure--Zmacs--Lisp--Output Configuration,
Realization Class--Zmacs--Lisp--Output Configuration.

Each configuration consists of a subset of the panes described above, possibly a Zmacs buffer, the mouse documentation line, and the status line. The mouse documentation line shows the operations which different mouse clicks can invoke at a given time, based on what the mouse is positioned over. The status line gives information about the state of the machine such as who is currently logged in, the time, the current package, the process run state, and file or network server information. The status line is a good place to check when you are experimenting with Mumble. For example, the current package might be some package you are not familiar with or the process run state might be OUTPUT HOLD - in which case some process is waiting to type out to a window which is not exposed. See the Symbolics or TI Explorer manuals for further information on the mouse documentation line or the status line. In this section, we give a screen shot and description of each configuration.

Demo Configuration

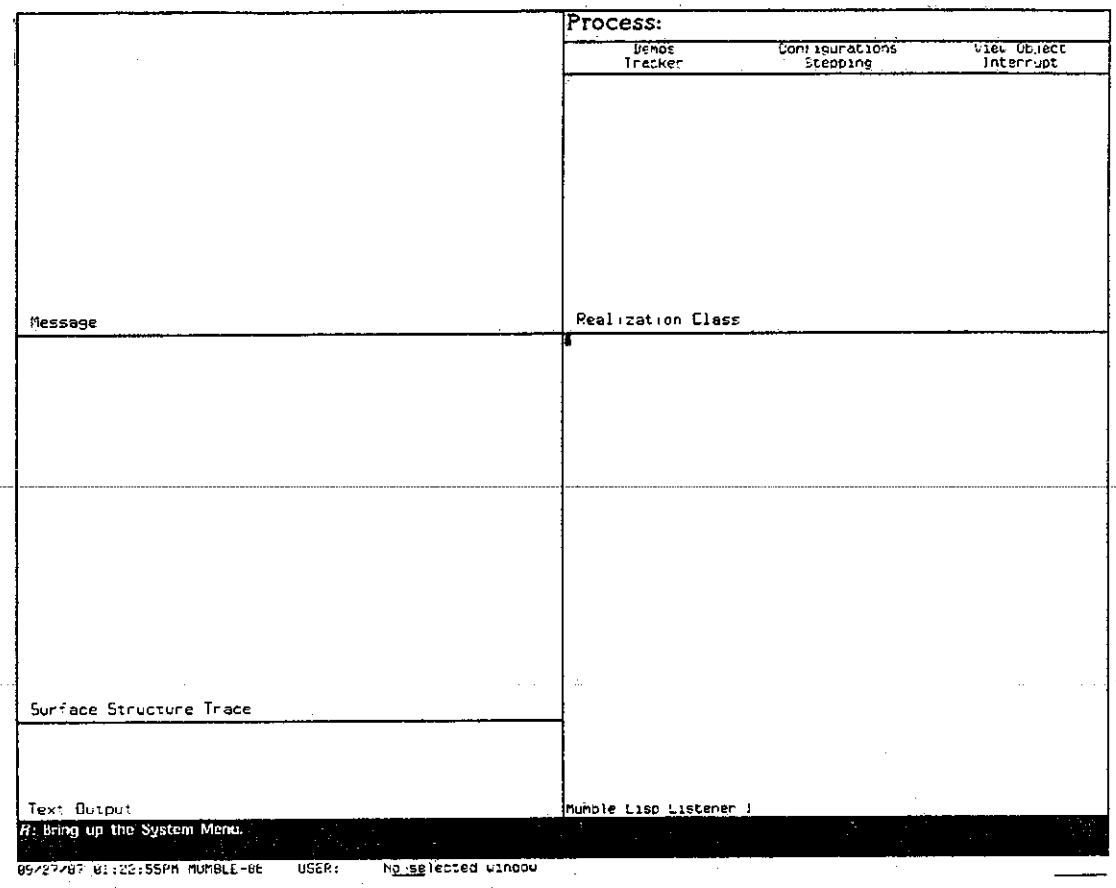


FIGURE 5.3

The Demo Configuration is shown in Figure 5.3. Moving clockwise from the upper right, this configuration contains the Process Pane, the Command Menu, the Realization Class Window, a Lisp Listener, the Text Output Window, the Surface Structure, and the Message Window. This configuration contains the most panes and will give you the most information about the realization of the input message to output text. It will show you the input message, the intermediate representation of the message, and the output text.

The three panes on the left-hand side of the screen (the Message Window, the Surface Structure Trace, and the Text Output Window) are placed in this top-to-bottom order to show the transformation of Mumble's input to its output. The top window (the Message Window) shows the input specification - the Realization Specification; the middle window (the Surface Structure Trace) shows the intermediate representation - the surface structure of the text; and the bottom window (the Text Output Window) shows the words of output text.

Zmacs--Lisp--Output Configuration

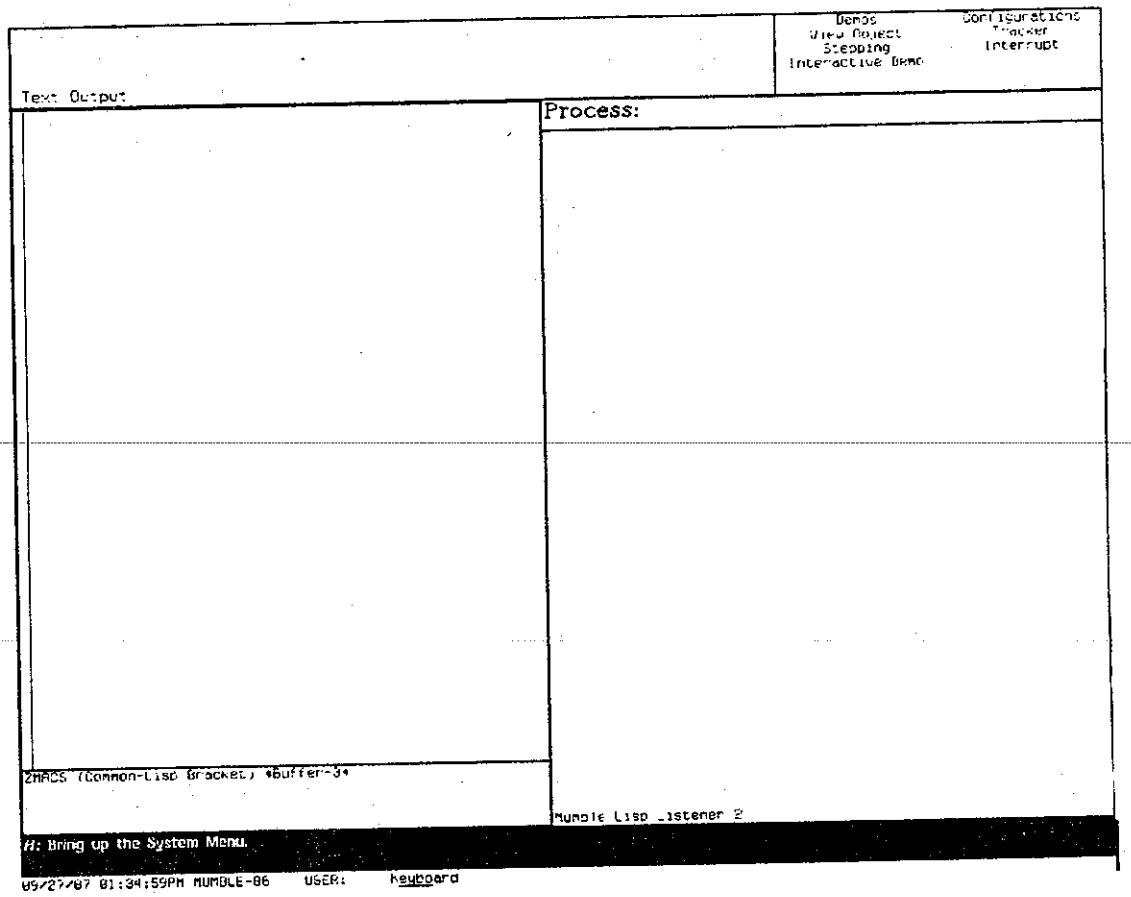


FIGURE 5.4

The Zmacs-Lisp-Output Configuration is shown in Figure 5.4. Beginning in the upper right-hand corner of the screen and moving clockwise, this configuration contains the Command Menu, a Lisp Listener, a Zmacs Editor, and the Text Output Window. This configuration is useful while editing and debugging Mumble's code.

Message--Zmacs--Lisp--Output Configuration

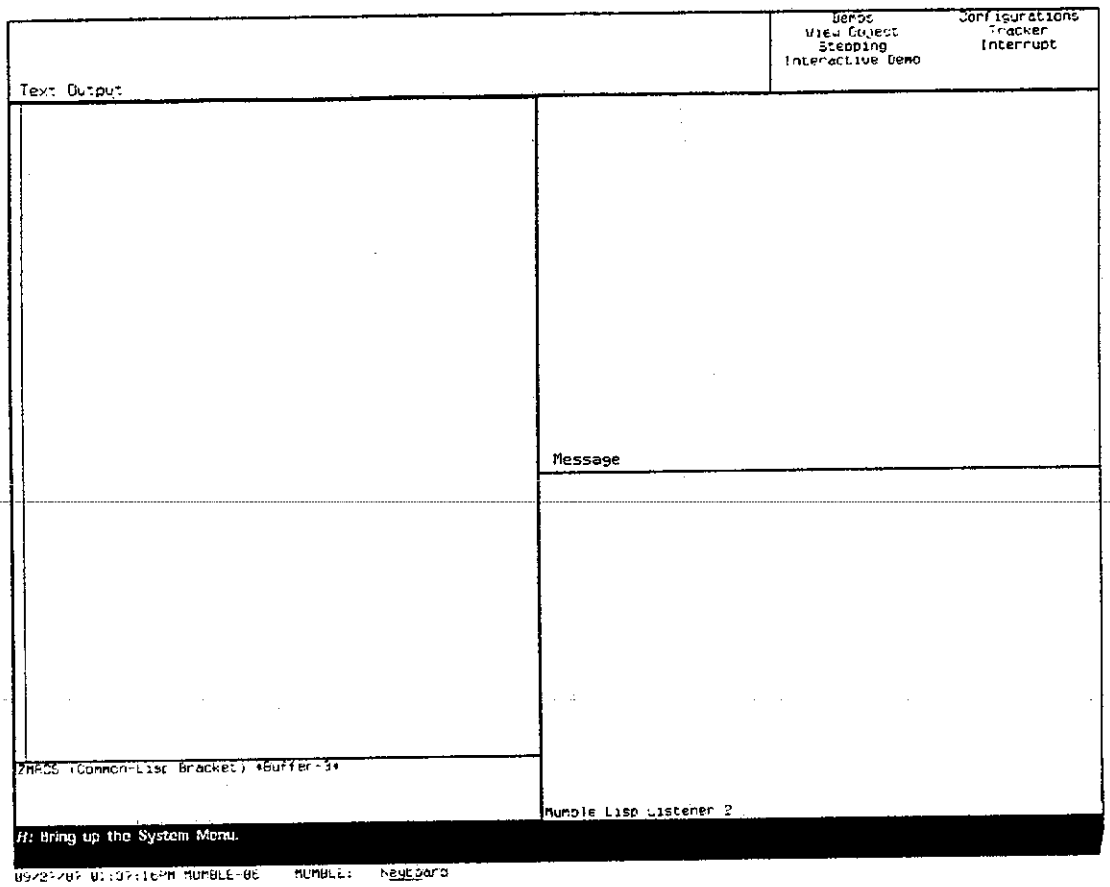


FIGURE 5.5

The Message-Zmacs-Lisp-Output Configuration is shown in Figure 5.5. Beginning in the upper right-hand corner of the screen and moving clockwise, this configuration contains the Command Menu, the Message Window, a Lisp Listener, a Zmacs Editor, and the Text Output Window. This configuration is useful if you are editing and want to see only the input message given to Mumble and not the intermediate representation of the message.

Surface Structure--Zmacs--Lisp--Message Configuration

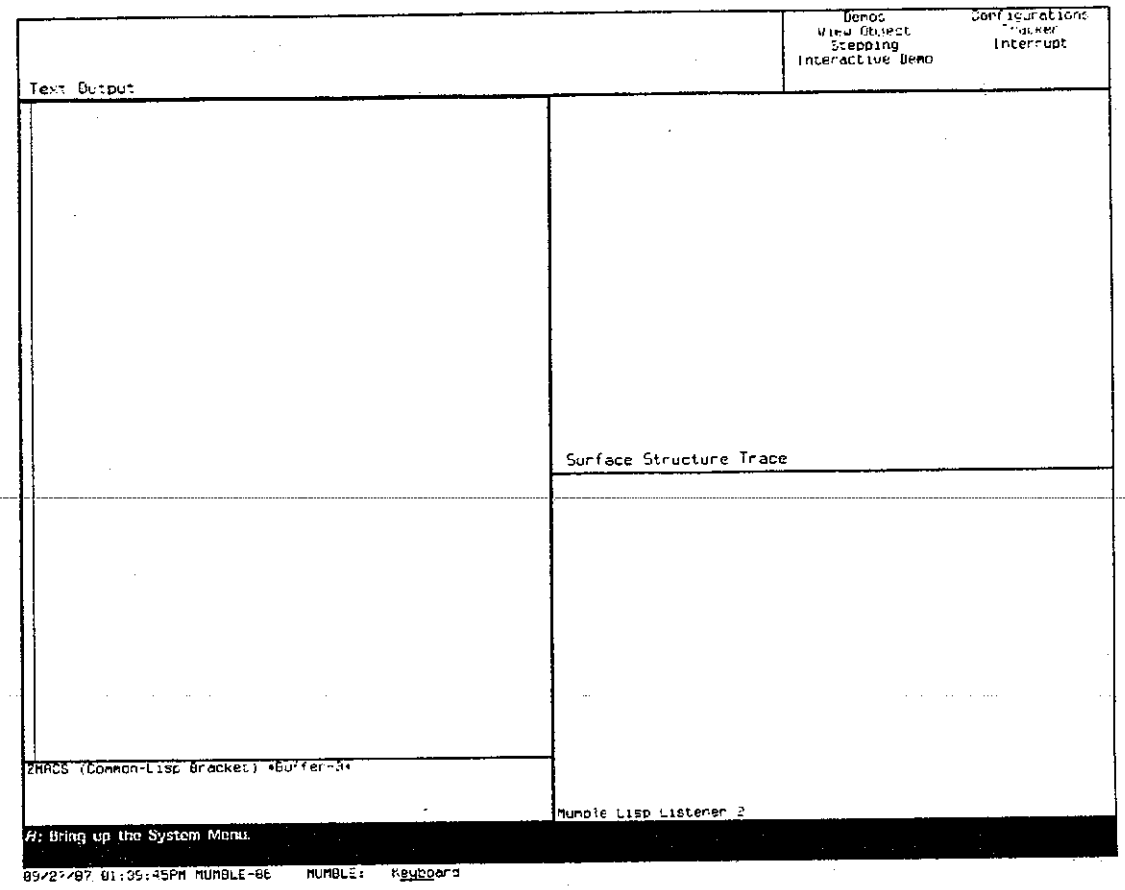


FIGURE 5.6

The Surface Structure--Zmacs--Lisp--Output Configuration is shown in Figure 5.6, Beginning in the upper right-hand corner of the screen and moving clockwise, this configuration contains the Command Menu, the Surface Structure Trace, a Lisp Listener, a Zmacs Editor, and the Text Output Window. This configuration is useful if you are editing and want to see only the intermediate representation of the input message and not the input message itself.

Realization Class--Zmacs--Lisp--Output Configuration

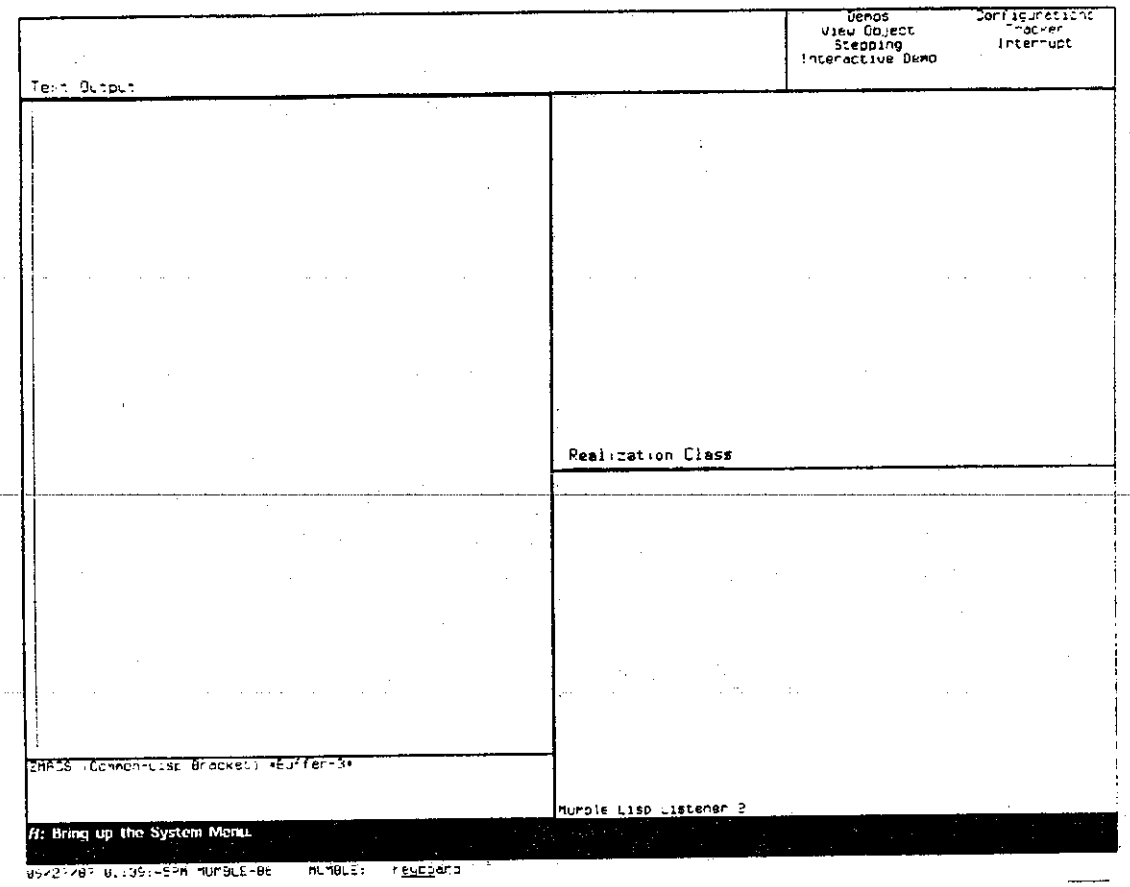


FIGURE 5.7

The Realization Class--Zmacs--Lisp--Output Configuration is shown in Figure 5.7. Beginning in the upper right-hand corner of the screen and moving clockwise, this configuration contains the Command Menu, the Realization Class Window, a Zmacs Editor, and the Text Output Window. This configuration is useful if you are editing and want to watch the Realization process during Mumble's execution.

5.4 DELIVERY AND INSTALLATION

In this section, we describe how to install the mumble files on your host system.

5.4.1 Installing Mumble on your system:

Installation involves two steps: reading the files from magnetic media, and customizing the files for your system.

Restoring from Magnetic Media:

MUMBLE can be distributed on tape cartridges in carry-tape or backup format for use on Symbolics or TI-Explorer Lisp Machines, or on industry-standard tape in backup format for use on DEC VAX machines running VMS. By special arrangement, we may also distribute bootable bands containing the Mumble system for use on TI-Explorer Lisp Machines.

SYMBOLICS: The procedures described below are valid for systems up to release 7.0.

If the tape is in backup format, physically load the tape in the tape drive and enter the file system editor. Select the "Local LMFS Operations" option. Another menu level will be displayed: you should select the "Read Backup Tape" option from it, and then "Reload all" and "Do it" from the menu displayed subsequently. You will need approximately 200 free blocks to load the files successfully. Once backup is completed, select the "Exit Level 2" option, and return to the lisp listener.

If the tape is in carry-tape format, see vol. 0 (Site Operations) of the Symbolics manual for help.

EXPLORERS: The procedures described below are valid for systems up to release 2.1.

Independent of the tape format, physically load the tape in the tape drive and enter the backup system (type SYSTEM-B).

If the tape is in backup format, select the "Prepare Tape" option, followed by the "Load Tape" option. (Note that backup tapes may only be restored from tape drives located on the machine you are logged in to; if this is not the case, you will receive an error message.) Finally, select the "Restore Directory" option. A menu will be displayed on which the "Do it" box should be selected.

If the tape is in carry-tape format, select the "Restore Symbolics Carry" option. From there you may specify the number and host name for the tape drive in which you have inserted the tape (see the Operations Manual for more details).

If a bootable system is included on the tape, you will receive special instructions with the tape. The procedure is essentially the same as installing a new Explorer system, however.

VAX:

Edit the following command procedure as needed and execute it:

$! allocate first available tape unit of either type (mt or mu):

$ allocate mt,mu tape

$! logically mount the tape -- the system will ask you to physically

$! mount the tape, and tell you where:

$ mount tape/override=identification/foreign

$! now copy the tape contents to disk, just sit back and watch the

$! pretty lights!!:

$ backup tape:mumble.backup YOUR$DEVICE:[YOUR-ROOT...]

$! ten minutes later, take the tape off and store it:

$ dismount/unload

5.4.2 Customizing the files

The system must now be customized for your site. Two different procedures are followed, depending on whether you are using a lisp machine or a vax.

SYMBOLICS AND EXPLORER:

To customize Mumble on these machines, you must change all the references to a specific host ("grumpy" at Umass) to your own host and indicate your root directory. There are only three places where these changes need to be made:

1) In LISPM-INIT (top level Mumble-86) change the pathname for the BOOTSTRAP file to reflect your host and root.

2) In BOOTSTRAP (top level Mumble-86) update the logical pathname assignment to reflect local machine names and/or directory structure (see the documentation on the file system local to your system for help on this). See below:

Symbolics:

```
(fs:set-logical-pathname-host
  "mumble-86"
  :translations '(("mumble-86:top;*.*.*" "YOUR-MACHINE:>YOUR-ROOT>*.*.*")
                  ("mumble-86:**;*.*.*"    "YOUR-MACHINE:>YOUR-ROOT>*.*.*")))
```

Explorer

```
(fs:add-logical-pathname-host
  "mumble-86"    "YOUR-MACHINE"
  '((top         "YOUR-ROOT;")
    (browser         "YOUR-ROOT.browser;")
    (grammar        "YOUR-ROOT.grammar;")
    ...
    (types              "YOUR-ROOT.types;")))
```

3) In VERSION-DB (top level Mumble-86), change specific references to host and root.

VAX:

Simply define the rooted logical MUMBLE$DISK, as follows:

```
$ define/exec/tran=(conc,term) mumble$disk YOUR-DEVICE:[YOUR-ROOT.]
$
```

6. INTERFACING TO MUMBLE

Contents

As we have discussed, Mumble-86 is a linguistic component for natural language generation. It is designed to be used in conjunction with a underlying applications programs responsible for initiating the generation process and determining the goals the utterances are to achieve. A planning component would be responsible for selecting the information to be communicated, determining what perspectives and rhetorical organization the information should be given, and choosing a mapping for the information onto the linguistic resources that the language provides. (For a more complete discussion of the generation process, see Sections 1.2 and 1.3.)

In part one of this section we discuss how one would build an interface between a simple underlying program and Mumble-86. In order to focus on interface issues, we assume minimal planning capabilities and focus on how model level objects can be linked to templates which build input specifications to Mumble.

In part two, we look at how Mumble can be run by itself, without an underlying program behind it. Our "stand alone" interface is used to test grammatical constructions and run demonstrations of Mumble alone. We discuss the syntax of the input specification language and how to build pre-defined "demos".

6.1 Providing an interface between an underlying program and Mumble-86

In the simplest conception of how an interface can work, the underlying program will already have a representation of what it wants to say -- some structured object or other internal expression -- and will want Mumble to take that object and convert it to reasonably fluent English.

To be concrete, suppose the object is a proposition that might be printed out as the expression **next-to(stove, refridgerator)** --- the assertion that a certain relation holds between two objects. Typically the relationship will hold among other

objects, and the objects will be involved in other relationships. Consequently we will want all three items to have their own, independent means of being rendered into English, and treat the proposition as a composition rather than a unique entity.

Given that breakdown, on Mumble's side of the interface three realization specifications will be assembled. For instance, one specifying a set of copular locative clauses using the words "*next to*", and two others specifying definite noun phrases using the words "*stove*" and "*refridgerator*" -- obvious choices perhaps, but simple underlying programs are often coded that deliberately.

The simplest basis for the interface is a table-lookup---a function that maps an object to its designated realization specification. To create the table in the first place there is another function, executed off-line as part of the process of defining the interface, that links the specific data structure that is the internal representation of each of the objects to a specific "template" whose operations will construct and return the appropriate realization specification.

To use Mumble, the underlying program in this simple example takes its data structure for the proposition and applies an instantiation function to it recursively to look up and construct realization specifications for it and the items it is composed of, then passes the specification for the proposition to Mumble as the argument to its toplevel function.

This example is loosely based on an early version of the representation we use in the program APT, a system for exploring text planning issues in the task of describing the layout of an apartment. We present some example objects, show how the interface can link them to realization specifications, show the templates that they use, and sketch a function that uses that apparatus to set up a call to Mumble.

6.1.1 Objects in the underlying program

Before we can speak concretely about any interface between Mumble and other programs, we have to look carefully at the the specific way that the underlying program represents and (crucially) implements the objects that it wants rendered into English. There are any number of ways that a proposition that is presented in print as **next-to(stove, refridgerator)** might be implemented in a Lisp program. In the mid-seventies the norm would have been to use list structure and symbols; today it is more likely to be instances of Flavor objects. For the sake of simplicity, let us take a middle ground and assume that the implementation uses Defstruct.

All that is absolutely required by the interface is that each item for which a mapping to realization specifications is defined must be implemented as its own, unique, first-class Lisp object -- otherwise there would be no way to pass the object as an argument to the instantiation function or give it an entry in the table.

Besides their nature as data structures, there is also the question of how the relation and the objects have been conceptualized. What are the types, and how are

tokens individuated and categorized? The design might be as simple as having one Defstruct definition for all "objects" and another for all "relations" and have individuals distinguished by the values of their "name" fields, or the design might involve an elaborate taxonomic hierarchy with distinctions between "definitional" and "assertional" knowledge and elaborate methods for distributing the definition of an item's properties by using inheritance and defaults.

The interface to Mumble is agnostic about such design decisions. This is easily achieved because all interface activities happen *before* Mumble is actually called (i.e. the realization specifications are created by the underlying program or its text planner, not by Mumble); consequently the underlying program's own access functions and type predicates can be used freely to decode the particular design in use.

To again keep this example simple, let us assume that objects like **refridgerator** are just instances of a single type, "object", implemented by Defstruct, and that their only property is their name, implemented by a print-name field. To make refering to the objects easy, we bind them to conveniently named variables.

```
(defstruct object (:print "#<object ~a>" (print-name object))
   print-name )
(defvar refridgerator (make-object :print-name "refridgerator"))
(defvar stove (make-object :print-name "stove"))
```

As for relations, it is probably simplest here to make each one a type of its own. This way, instantiating the relation with the Defstruct "make-" function will provide a unique lisp object to serve as the representation of the proposition. Of course this does not provide a first class object for the relation itself, but that will not be a problem in this example since we will only use relations as parts of propositions and never say anthing about them. (I.e. we won't predicate anything of a relation or modify it, e.g. "*right next to*".)

```
(defstruct next-to (:print "#<next-to ~a ~a>"
                           (prime next-to)
                           (secondary next-to))
  prime
  secondary )
(defvar the-refridgerator-is-next-to-the-stove
        (make-next-to  :prime stove  :secondary refridgerator ))
```

6.1.2 Templates for the realization specifications

Realization specifications are the only information that Mumble receives about the objects in the underlying program that it has been given to render into English. Like all the other data types used by Mumble, they are structured objects, and the definition of the values that their fields can take on is relatively complex. Like the nodes and slots that make up the surface structure, realization specifications are temporary objects that are contructed as needed and forgotten as soon as they have been used. Each specification refers to a single, specific instance of an object; if the

same object appears multiple times (as in *Guisseppi shaved himself*) there will be a different realization specification for each, even though the specifications will typically have identical values in their fields.

For these reasons---and also to provide a representational vocabulary special to the message level, with its own possibilties for generalizations---the underlying program and its text planner do not work directly with realization specifications but with specification "templates": permanent, parameterized functions that return realization specifications when executed.

Just as there is a predefined set of phrases and position labels to define the properties of possible surface structures, there is predefined set of specification templates, and just as the computational linguist can extend Mumble's grammar by using special forms to define more labels or phrases, there is a special form for defining more specification templates. An example of a specification template definition appears in section 6.1.4.

6.1.3 The table linking objects to templates

Specification templates are parameterized functions that construct and return realization specifications---in effect they are the intensional specification "types" to the extensional realization specification "tokens" that are instantiated and used as needed. To take advantage of the generality, the association between an object and its specification is mediated by an expression---its value in the interface table---that is analogous to a function call: the application of a specification template to a set of arguments.

As an example, consider the two objects in our example, **refridgerator** and **stove**. They will use the same template, though of course with different arguments; this template, Instance-of-a-kind, dictates that the objects will be realized by noun phrases (or under certain grammatically controlled circumstances by a pronoun), but does not constrain the common noun that will be used (a reference to a conceptual "kind") or the determiner, leaving these to be spelled out by the arguments.

```
(define-default-specification  apt:refridgerator
   :template-name  instance-of-a-kind
   :arguments  ("refridgerator"
                'third
                'singular
                'neuter
                'always-definite ))

(define-default-specification  apt:stove
   :template-name  instance-of-a-kind
   :arguments  ("stove"
                'third
                'singular
                'neuter
                'always-definite ))
```

Define-default-specification associates an object from the underying program or the text planner (i.e., its first argument -- which is evaluated), with an "entry" in a table maintained just for this purpose (the Object-to-specification-table-entry object type, see Section 3.1.1). To retrieve the entry, the access function "Default-specification" is provided. Thus we could have the following transactions in a lisp-listener. Type-in by the designer is given in bold.

```
>
>apt:refridgerator
>#<object refridgerator>
>
>(default-specification  apt:refridgerator)
>#<table-entry-for-individuals #<object refridgerator>>
>
>(describe  *)
>#<table-entry-for-individuals #<object refridgerator>> is a TABLE-ENTRY-
FOR-INDIVIDUALS
        name: |#<object refridgerator>|
        object-assigned: #<object refridgerator>
        schema: #<specification-schema specific-individual
        arguments: (  #<word refridgerator>
                      third
                      singular
                      neuter
                      always-definite )
#<table-entry-for-individuals #<object refridgerator>> is implemented as
an ART-Q type array.  It uses %ARRAY-DISPATCH-WORD; it is 5 elements long.
#<table-entry-for-individuals #<object refridgerator>>
>
```

6.1.4 Table entries for classes rather than individuals

If only because they involved mapping to NPs with different common nouns, the refridgerator and stove required their own entries. The **next-to** relation on the other hand is going to use the same constructions and phrasings regardless of the objects being related: there would be little point to having a separate entry for each instance of this relation such as **next-to (stove, refridgerator)**. To accomodate this need, we have a variation on the Define-default-specification special form, and use it in the way that one would expect:

```
(define-default-class-specification  'next-to
   :template-name  locative-relation
   :arguments ( (prime self)
                (secondary self) ))
```

This form differs from the form for individual objects in only one crucial respect, namely that the expressions used in specifying the arguments are intended to be evaluated at run-time rather than decoded when the form is postprocessed. The arguments will normally be given as access functions that extract embedded objects

from the data structure that makes up the object being realized; the variable "self" is bound to that object by the instantiating routine for just this purpose.

6.1.5 A brief look at templates

A specification template is simply a body of lisp code that constructs and returns a realization specification. As such, it is a topic better discussed with the details of the specification language (see Sections 2.2 and 6.2), though for the sake of local completeness we can go through one example. Below is the template used for the two objects in our example. The special form that defines it is essentially the equivalent of the regular lisp "defun".

```
(define-specification-template instance-of-a-kind
                               (kind-term
                                person number gender
                                determiner-policy)
  (let ((b (make-a-bundle 'general-np)))
    (set-bundle-head b (typecase kind-term
                         (word (make-a-kernel 'np-common-noun
                                              kind-term))
                         (otherwise
                            (instantiate-mapping kind-term))))
    (add-accessory b :person person)
    (add-accessory b :number number)
    (add-accessory b :gender gender)
    (add-accessory b :determiner-policy determiner-policy)
    b))
```

Discussion of "bundles", "kernels", or "accessories" must be reserved for section 2.2. One thing to note, however, is that this template is prepared to accept either a word or some instantiate-able object as its "kind-term". This points up the tentative, and somewhat arbitrary, design that templates have today: On the one hand they will embody a model of the semantic structure of English (according to their author's view of course), e.g. that noun phrases can be assembled from the specialization of a concept that represents an entire class (the source of the kind-term). At the same time they "cut corners" to expedite their use with underlying programs that don't support so thorough a model (e.g. allowing a common noun in place of a concept).

Templates are not part of Mumble-86.[1] They are part of our ongoing experiments with text planning, where they are candidates for the planner's output vocabulary. We include them in what we distribute because we have found them to be very useful (they are in effect an alternative, very compact, notation for realization specifications), and because they can be used to establish a very clear, if expedient, means of using objects in the underlying applications program directly as the input to Mumble.

6.1.6 Mapping objects to specifications

We should presume that the interface is fully defined before it is used, i.e. all objects (or classes of objects) that will ever need realization specifications will have already been given entries in the object-to-specification-table. However since the interface sits on the text planner's side of the fence, as it were, there is nothing to keep a designer from constructing entries on the fly (or for that matter doing without default entries entirely), but in our experience we have found prior definition has always been simplest in the end.

In any event, a single function has been designated as the conventional way to map objects into their specifications, **instantiate-mapping**, which is used by all of the templates that we distribute when processing embedded elements. Instantiate-mapping checks for the existence a default specification for either the object as an individual or as a member of a class, and includes a "hook" for an interface designer

[1] One must draw the line somewhere. If we could imagine mapping the 1979 version of Mumble into present terms (i.e. the version that is written up in McDonald, 1980, 1983; see section 1.4), these templates would be the first part of a clear rationalization of the capabilities that were inherent in the Lisp-based "dictionary entries" of the time. Those entries were unformalizable, much too powerful and wide-ranging in what they did to permit any ready understanding of what they were capable of or where their generalizations lay. Such unconstrained and unteachable facilities have been deliberately removed from Mumble-86. This has often diminished its intrinsic competence, but is in keeping with our present view of it as just one individual component in a larger generation system.

to incorporate their own, idiosyncratic means of associating objects with the processes that will construct their realization specification.

```
(defun INSTANTIATE-MAPPING  (object)
  (or (let ((table-entry (default-specification object)))
        (if table-entry
            (instantiate-table-entry table-entry object)))
      (let ((class (member-of-a-class? object)))
        (if class
            (let ((class-table-entry (default-specification class)))
              (if class-table-entry
                  (instantiate-class-table-entry class-table-entry
                                                 object)))))
      (when application-specific-mapping-algorithm
        (computed-mapping object))))

(defvar application-specific-mapping-algorithm  nil
  "set to the name of the appropriate function by an application")

(defun COMPUTED-MAPPING (object)
  (let ((result-of-mapping
          (funcall application-specific-mapping-algorithm object)))
    (etypecase result-of-mapping
      (cons (let ((template  (car result-of-mapping))
                  (arguments (cdr result-of-mapping)))
              (apply-template template arguments)))
      (specification  result-of-mapping))))
```

The hook is, of course, the global variable Application-specific-mapping-algorithm, which can be set to any arbitrary function choosen by the designer of an application-specific interface. As may be clear from the code of Computed-mapping, the function that manages this specific interface, the designer has the option of constructing a specification by whatever means it chooses, or can encapsulate and share much of the work by construing its effort as just the search for the appropriate template.

For more examples and details of interfaces between Mumble-86 and underlying applications programs, and of how we expect them to evolve, the reader is referred to one of our recent papers: "From Water to Wine: Generating Natural Language Text from Today's Applications Programs", McDonald & Meteer 1987.

6.2 THE "STAND ALONE" INTERFACE

The stand alone interface allows the user to define "demos" which run Mumble without an underlying program or planner behind it. Essentially, the input specifications are defined as merely symbols and lists adhering to a particular syntax. CREATE-MESSAGE then parses this representation and creates the actual specifications Mumble requires as input.

In addition to allowing the user to define specificatons directly, this interface also allows the use of specification-templates (see section 6.1.2), which return realization specifications in the same manner as if they were employed by a planner.

6.2.1 The form of stand alone specifications

In this section we describe the syntax of stand alone specifications and show an example in Figure 6.1.

1. Kernel Specifications are denoted by lists: (name &rest arguments). NAME is the name of the realization function to use, which must be the name of a realization-class, curried-realization-class, or single-choice. Arguments can be any specification, including words, pronouns, kernels, and bundles, or templates. The number of arguments must match the number of parameters of the realization function.
2. Bundle Specifications are also denoted by lists: (bundle-type &key head accessories further-specifications). They can be distinguished from Kernels because the car must be the name of a bundle-type, such as discourse-unit or general-np. Depending on the bundle-type, these keyword arguments may be semantically optional. They are all optional as far as the syntax of the standalone interface cares. Because of the flat property-list style of these, the accessories and the further specifications must be encloses in parentheses. For example:

```
(general-np :head <spec>
            :accessories (:number singular :gender male ...)
            :further-specifications (fspec1 fspec2 ...))
```

3. Further Specifications are sublists of the further-specifications list of a bundle. Their syntax is pure plist: (&key specification attachment-function). The specification may be any type of specification or a template name plus arguments and the attachment function must be the name of an attachment class or an attachment point. (The attachment function is optional on discourse unit bundles.)
4. Template applictions are also denoted by lists: (template-name &rest arguments). The template-name must be the name of a predefined template and the number of arguments must match the number of parameters in the template. The arguments

may be any specification, including words, pronouns, kernels, and bundles, or templates

5. Words are denoted by strings: "Soleil". They must be already defined as objects of type word in Mumble's grammar.
6. Pronouns are denoted by symbols: first-person-singular. They also must be pre-defined as objects of type pronoun in the grammar.
7. In addition, the standalone interface provides two specially interpreted symbols: *self* and *head*. While translating the further specifications of a bundle, these symbols are bound to the bundle itself and to the head of the bundle, respectively. This is done because s-expressions are trees, and therefore you cannot have sublists refer to the list itself or parts of the list.

The example in Figure 6.1 shows a stand alone specification which combines direct definition of parts of the specification with the use of templates to build other parts. Specifications which are defined directly are turned into actual specifications by the function CREATE-MESSAGE (see Section 6.2.2). The bundles DISCOURSE-UNIT and GENERAL-CLAUSE are defined in this way in the demo: their heads are specified and, in the case of general-clause, the accessories are assigned. (Neither bundle has further-specifications.) In contrast, the arguments to the kernel chase, "(named-object ...)" and "(instanceof-a-kind...)" both use templates to build the bundles and specify the parts.

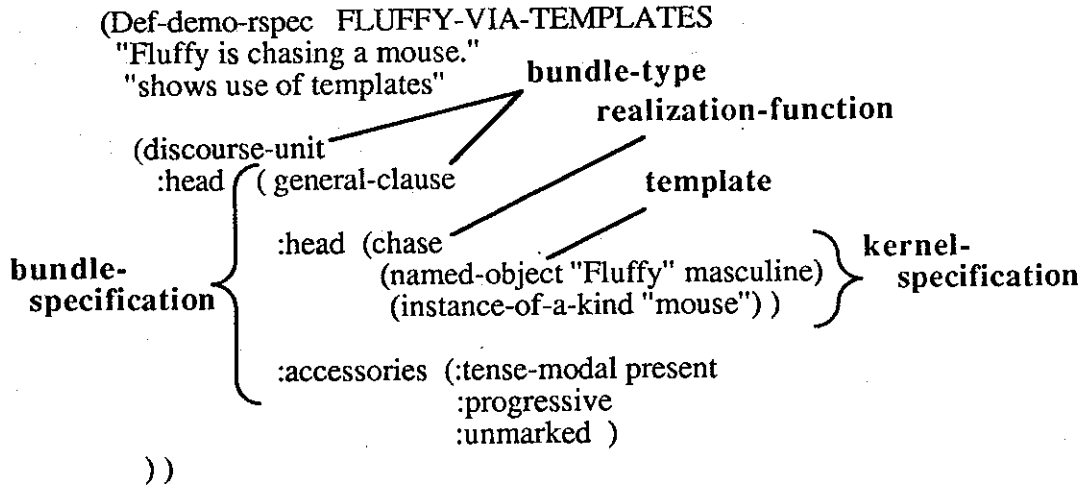


FIGURE 6.1

Figure 6.2 shows the template NAMED-OBJECT. It builds an NP-BUNDLE, builds the appropriate kernel for the head, and sets the accessories PERSON, GENDER, NUMBER, and DETERMINER-POLICY using predefined operators. (For

contrast, see Figure 2.4, which shows a stand alone specification with all parts defined directly rather than using templates.)

```
(define-specification-template NAMED-OBJECT (name gender)
  (let ((b (general-np)))
    (set-bundle-head b (if (specificationp name)
                           name
                           (Make-a-kernel 'np-proper-name name)))
    (third-person b)
    (set-gender b gender)
    (singular b)
    (no-determiner b)
    b))
```

FIGURE 6.2

The form of the demo specifications is very flexible as to what is defined directly and what uses templates. Figure 6.3 shows the same specification defined this time using a template to build the bundle GENERAL-CLAUSE (shown in Figure 6.4).

```
(Def-demo-rspec FLUFFY-VIA-TEMPLATES-2
  "Fluffy is chasing a mouse."
  "shows use of templates with default specifications"
  (discourse-unit
      :head (ongoing-process
              (chase :Fluffy :mouse1) )
  ))
```

template

arguments to the template

"objects" with default specification mappings

FIGURE 6.3

```
(define-specification-template ONGOING-PROCESS
                               (event)
  (let ((k event)
        (b (make-a-bundle 'general-clause)))
    (set-bundle-head b k)
    (present-tense b)
    (progressive b)
    (unmarked-clause b)
    b ))
```

FIGURE 6.4

This example also takes advantage of "default specifications". In section 6.1 we discussed how objects in an underlying program could be linked to templates. In this example, we have (previously) linked the symbol ":Fluffy" to a default-specification (shown in Figure 6.5) which, when executed, invokes the template named-object with the appropriate arguments. (Note that the practice of linking symbols to default specifications may be dangerous, since they are so easily overloaded.)

```
(define-default-specification :Fluffy
        :template-name  named-object
        :arguments ( "Fluffy" masculine ))
```

FIGURE 6.5

6.2.2 Creating Specifications

The function CREATE-MESSAGE (shown below) simply maps through a list, turning each element into a specification. The function TRANSLATE-SPECIFICATION (also shown below) does the actual work of determining how each element should be translated. In some cases, this can be determined by the type of the element, for example a string is turned into a Mumble WORD object. When the element is itself is a list, the translation can be determined by looking at the first element in the list: if it is a bundle type, the element should be turned into a bundle, if it is a realization function, it should be turned into a kernel, if it is a list then translate-specification should be called recursively. Note that when first element is ambiguous (e.g. it both names a template and a realization function) the order of the cond clauses in TRANSLATE-SPECIFICATION defines an implicit preference for one translation method or another. (the template will be applied, rather than a kernel built).

```
(defun CREATE-MESSAGE  (list-of-rspec-sexps)
  "the main function to translate a demo sexp into a real message."
  (setq *rspec-alist* nil)
  (mapcar #'translate-specification list-of-rspec-sexps))
```

```
(defun TRANSLATE-SPECIFICATION (spec)
  (declare (special current-bundle))
  (flet ((list-of-specs? (s)
           (and (listp s) (listp (car s))))
         (bundle? (s)
           (and (consp s) (symbolp (car s))
                (mumble-value (car s) 'bundle-type)))
         (template? (s)
           (and (consp s) (symbolp (car s))
                (mumble-value (car s) 'specification-template))) )

    (cond ((list-of-specs? spec)
           (mapcar  #'translate-specification spec))
          ((eq spec '*head*)
           (head current-bundle))
          ((eq spec '*self*)
            current-bundle)
          ((stringp spec)
           (gethash spec pnames-to-words))
          ((symbolp spec)
           (if (keywordp spec)
               (instantiate-specification-mapping spec)
               (pronoun-named spec)))
          ((earlier-mention spec))
          ((template? spec)
           (translate-template-application spec))
          ((bundle? spec)
           (translate-bundle-specification spec))
          ((consp spec)
           (translate-kernel-specification spec))
          (t
                  (mbug "~s cannot be translated into a specification."
spec)))
      ))
```

We show the functions translate-kernel-specification and translate-bundle-specification below to give you a flavor of the translation processes.

```
(defun TRANSLATE-KERNEL-SPECIFICATION (kernel-sexp)
  (dbind (rfun-name &rest args) kernel-sexp
    (let ((rspec  (make-kernel-specification
                    :realization-function
                       (translate-realization-function rfun-name)
                    :arguments
                       (mapcar #'translate-specification args))))
      (add-mention kernel-sexp rspec)
      rspec)))
```

```
(defun TRANSLATE-BUNDLE-SPECIFICATION (bundle-sexp)
  (dbind
    (name &key head accessories further-specifications)
    bundle-sexp
    (let ((btype       (bundle-type-named name))
          (head-spec   (translate-specification head)))
      (bind ((current-bundle    (make-bundle-specification
                                   :bundle-type  btype
                                   :head   head-spec)))
        (add-mention bundle-sexp current-bundle)
        (when accessories
          (setf (accessories current-bundle)
                (order-accessories accessories btype)))
        (when further-specifications
          (setf (further-specifications current-bundle)
                (mapcar #'translate-further-specification
                        further-specifications)))
        current-bundle))))
```

6.2.3 Building Demos

The demo defining forms allow the user to predefine a demo and have it appear on the list under the DEMO option in the Mumble window configuration. The name of the demo is also defined as a function name, so that it can be called from the lisp-listener.

DEF-DEMO-RSPEC (shown below) defines a function which is the demo. That function calls MUMBLE-A-DEMO which creates a message out of the RSPEC-LIST as the first step (which means we always get fresh specifications even when a demo is run more than once). It also handles initialization and clearing the windows in the Mumble configuration. Finally, it calls the top level function in Mumble: MUMBLE.

```
(defmacro DEF-DEMO-RSPEC (demo-name
                          text-produced
                          reason-for-this-demo
                          &body rspec-list)
  (check-type demo-name            symbol)
  (check-type text-produced        string)
  (check-type reason-for-this-demo string)
  `(progn (pushnew '(,demo-name
                     :value
                     ,demo-name
                     :documentation
                        ,(format nil "~a~%~a" text-produced reason-for-this-
demo))
                      *demo-choice-item-list*
                      :key #'car)
          (add-association! ',demo-name ',rspec-list *demo-sexps*)
          (record-source-file-name ',demo-name 'def-demo-rspec)
          (defun ,demo-name ()
             (mumble-a-demo ',rspec-list))))
```

```
(defun MUMBLE-A-DEMO (message)
  (setq pending-discourse-units
        (create-message  message))
  (initialize-mumble)
  (when *window-code?*
     (initialize-message-window-and-display-message pending-discourse-units))
  (mumble pending-discourse-units))
```

DEF-DEMO-RSPEC is only for defining demos from stand alone specifications. However, there are other special forms which allow you to predefine demos which use model level objects or call a planner.

```
(defmacro DEF-DEMO-MODEL-OBJECT (demo-name exp-whose-value-is-the-object)
  "defines a function and enters it on the item list used by the command menu
   demo option."
  `(progn (push ',demo-name  *demo-choice-item-list*)
          (record-source-file-name ',demo-name 'def-demo-model-object)
          (defun ,demo-name ()
          (mumble-a-model-object ,exp-whose-value-is-the-object))))

(defun MUMBLE-A-MODEL-OBJECT (model-object)
  (initialize-mumble)
  (initialize-message-window-and-display-message pending-discourse-units)
  (let ((realization-specification
          (instantiate-default-specification model-object)))
    (check-type realization-specification specification)
    (mumble (list realization-specification))))


(defmacro DEF-DEMO-PLANNER-RUN (demo-name form-that-is-a-call-to-a-planner)
  "defines a function and enters it on the item list used by the command menu
   demo option."
  `(progn (push ',demo-name *demo-choice-item-list*)
          (record-source-file-name ',demo-name 'def-demo-planner-run)
           (defun ,demo-name ()
           (mumble-a-planner-run ',form-that-is-a-call-to-a-planner))))

(defun MUMBLE-A-PLANNER-RUN (form-that-is-a-call-to-a-planner)
  (initialize-mumble)
  (initialize-message-window-and-display-message pending-discourse-units)
  (eval form-that-is-a-call-to-a-planner))
```

7. THE INTERACTIVE DEMO

Contents

The Interactive Demo is used to change a predefined Mumble input message, a "demo", and to see the effects those changes have on Mumble's output. It is possible to add, delete, and replace the parts of a Mumble input message. At any time, the user is able to get help information and to view Mumble objects. There are also options to run and save an input message.

This facility was designed to help new users of the system understand it better by seeing for themselves the relations between input specifications and the resulting output text. It is also useful for developers to test new grammatical constructions as it is easy to vary the input specification to test the construction in a variety of contexts. Since by its very nature the Interactive Demo assumes no underlying program behind Mumble (the user acts as the underlying program), it uses our "stand alone" interface, which we discuss in detail in Section 6.2. We discuss the particulars of the input specification language in Section 2.2.

The Interactive Demo enforces the syntax of an input message, not the semantics. Therefore, when editing the input message, it is easy to come up with a message which, when given to Mumble, will result in an error. The error may be that the output produced by Mumble is syntactically incorrect or that the code produces an error message because the construction of the input message is not one which Mumble expects. Part of the process of using the Interactive Demo is to learn what kinds of message constructions work when given to Mumble and what kinds do not.

In Section 7.1, we give an overview of the Interactive Demo: how to start and exit, the screen configuration, and the command menu. In Section 7.2 , we present a detailed description of how to edit a message is given. In Section 7.3 we walk step by step through a sample editing session of a Mumble message.

7.1 Overview

To get to the Interactive Demo, select INTERACTIVE DEMO from the Mumble Command Menu in any of Mumble's screen configurations. To exit the Interactive Demo, select EXIT from the Interactive Demo Command Menu and you will be returned to the Mumble configuration from which the Interactive Demo was called.

The screen configuration for the Interactive Demo is shown in Figure 7.1 and contains the following panes:

MESSAGE WINDOW where the input message is printed much the same as in the Message window in Mumble's Demo Configuration;

TEXT OUTPUT WINDOW where Mumble's output is displayed, word by word, as Mumble produces it;

DOCUMENTATION WINDOW where help information is printed as theuser requests it;

LISP LISTENER where Mumble objects are printed and lisp commands are interpreted;

COMMAND MENU where the user can select different commands.

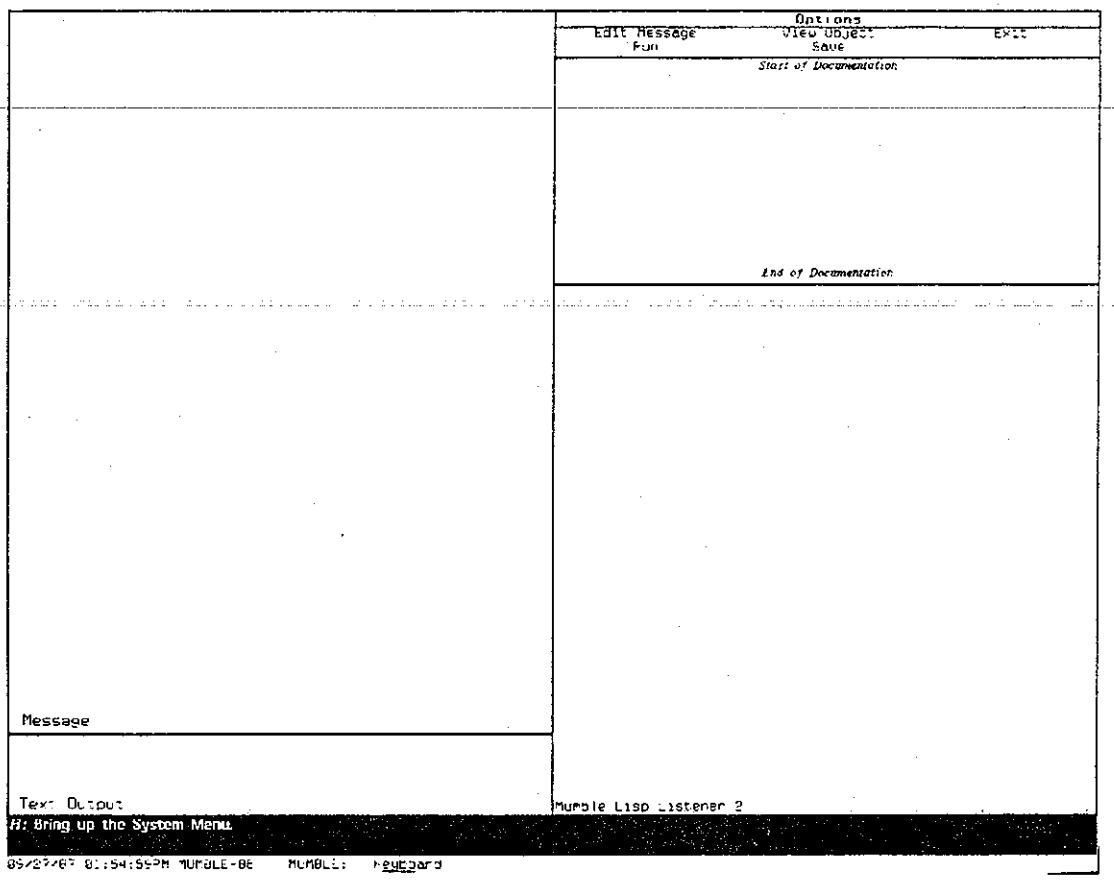


FIGURE 7.1 INTERACTIVE DEMO SCREEN CONFIGURATION

It is possible to scroll the text in the Message Window, the Documentation Window, and the Lisp Listener. To scroll text in the Message Window, slowly move the mouse to the left-hand side of the window until an up-down arrow appears. The documentation line will then give information about how to scroll the text depending upon which mouse button you click. Scrolling in the Lisp Listener is the same as scrolling in the Message Window. In the Documentation Window there

are two ways to scroll: left-hand side scrolling (which has already been described) and scrolling by bumping the mouse against the middle of the top or the bottom of the window.

The Interactive Demo Command Menu contains the following options: Edit Message, Run, View Object, Save, and Exit.

The **Edit Message** command displays a menu of the names of all the messages which have already been defined. Selecting one of the messages from the menu will print it in the Message Window, thus allowing you to edit it.

The **Run** command runs Mumble with the message currently displayed in the Message Window. Mumble's output will be displayed in the Text Output Window.

The **View Object** command allows you to see any Mumble object printed in the Lisp Listener. Selecting the VIEW OBJECT command displays a menu of all the Mumble object types. Selecting one of the object types brings up a menu of all the objects of that type which are currently defined. Selecting one of those objects will print that object in the Lisp Listener using the lisp "describe" routine. (If no objects of a particular type have been defined, a message will be displayed telling you so.)

The **Save** command saves the message currently displayed in the Message Window to the editor buffer, SAVED-MESSAGES. Note that the message is written to a buffer, not a file; the user can save the buffer to a file, if necessary.[1] Once the message has been written to the buffer, the user can evaluate the message, thus making that message one of the defined "demos". It will be available via the EDIT MESSAGE command in the Interactive Demo Command Menu or the Demos command in the Mumble Main Command Menu. The name of the demo will be TMP-MESSAGE *x* where *x* is a number. To change the name of a demo, go to the SAVED-MESSAGES buffer. The first line of any saved message will be:

```
(def-demo-rspec tmp-messagex
```

Replace TMP-MESSAGE *x* with the name of your choice.

The **Exit** command exits the Interactive Demo and returns the user to the Mumble screen configuration from which the Interactive Demo was called.

7.2 Editing a Message

To edit a message, first choose one of the defined messages by selecting the EDIT MESSAGE command from the Command Menu. After selecting one of these messages, it will be printed in the Message Window. You may select any piece of

[1] Messages are sent to the buffer as text only and thus are not formatted. You may "pretty print" the message with the META-CONTROL-Q keystroke.

the current input message by placing the mouse over that part of the message in the Message Window. (The documentation line at the bottom of the screen will say which piece of the message is currently selected.) Clicking on any mouse button while part of the message is outlined will bring up a menu of the options for that particular piece of the message. There are different sets of options for different pieces of the message. Currently, the following options have been defined: HELP, VIEW OBJECT, DELETE, REPLACE and ADD.

Mumble messages are made up of bundle specifications, kernel specifications, and further specifications. It is these specifications and their individual parts which you will be editing. Each of these specifications has required information which must be specified before you can run or save the current message. Bundle specifications contain a bundle head, accessories (which include an accessory type and possibly an accessory value), and further specifications. The bundle head and some of the accessories are required information for a bundle specification. Different types of bundles have different required accessories. Kernel specifications contain a realization function and some arguments. The realization function and the corresponding number of arguments are the required information for a kernel specification. (The number of arguments for a kernel specification is the number of parameters for the realization function of the kernel specification.) Further specifications contain an attachment function and a specification. Both of these pieces are required information for a further specification. For a more complete discussion of the input specification language, see Section 2.2.

The following list shows the possible operations for all the parts of a message. Only the options ADD, DELETE, and REPLACE are shown; HELP is always available, and VIEW OBJECT is available except when you are adding a new piece of the input message.

Bundle specification	Operation depends upon its position in message
Bundle Head	Add, Replace
Accessory Type	Add, Delete
Accessory Value	Add, Replace
Further Specification	Add, Delete
Attachment Function	Add, Replace
Specification	Add, Replace
Kernel specification	Operation depends upon its position in message
Realization Function	Add, Replace
Argument(s)	Add, Replace

Note that while editing, any time a menu is prompting you for information, you can simply move the mouse out of the menu and whatever operation you were currently performing will be canceled.

7.2.1 Command Menu Options

The **Help** option prints information in the Documentation Window. The type of information printed is usually some information about the current piece of the input message which is being edited and the different options available for that part of the message. The HELP option is available in every Options menu, whether it is an Options menu for a piece of the input message, or one for a placeholder for a piece of the message.

The **View Object** option prints, in the Lisp Listener, that part of the input message which is currently being edited. The object is printed using the lisp "describe" routine. The VIEW OBJECT option is only available in the Options menus for pieces of the input message, and not in Options menus for placeholders for a piece of the message.

The **Delete** option allows the user to delete pieces of the input message. Currently, only accessories and further specifications canbe deleted from the message.

The **Add** option allows the user to add a piece of the input message while the **Replace** option allows the user to replace a piece of the message with something which is syntactically, not necessarily semantically, equivalent to the piece which is currently being edited. We discuss exactly how to add and replace particular objects in section 7.2.2.

When both the Replace and Add options are available for a piece of the input message, the operations performed for these options are the same. However, the manner in which you get the Options menus will be different. When adding a piece of the input message, to get an Options menu you will outline placeholders for a piece of the message, such as !! ADD BUNDLE HEAD !!, !! ADD ATTACHMENT FUNCTION !!, or AGENT (where AGENT might be a placeholder for an argument in a kernel specification), and then click any mouse button. Then to add a piece of the input message you will select ADD from the Options menu. When REPLACING a piece of the input message, to get an Options menu, you will outline that piece of the message you wish to replace and click any mouse button. Then to replace that piece of the message, you will select REPLACE from the Options menu. For accessories and further specifications, you are able to add and delete them, but not to replace them.

When replacing or adding pieces of the input message, remember that Mumble will only enforce the syntax of the input message, so even if the Interactive Demo allows a particular addition or replacement, the edited message will not necessarily run correctly when given to Mumble; Mumble could either produce an error message or produce incoherent English. A general guideline to follow when replacing parts of the message is to replace like with like. For example, if you are replacing a word which is a noun it is best to replace that word with another word which is also a noun. And if you are replacing a realization function which is a curried realization

class, it is best to replace it with another curried realization class. Replacing something with an object of a different type may work, but it is not guaranteed to work.

7.2.2 Adding and replacing particular objects

The following sections describe how to add and replace, all the different types of specifications and their parts.

Specifications

When adding or replacing bundle or kernel specifications, it is possible to create a new specification or to use an existing specification. An existing specification is any specification which is already in the input message. When adding or replacing something with an existing specification, you will choose a specification from a menu which lists all the existing specifications. If you choose to add a new specification, then you must choose the type of specification: BUNDLE or KERNEL. If you choose to add a bundle specification, then you will be asked to select the bundle type: CONJUNCTION-BUNDLE, DISCOURSE-UNIT, GENERAL-CLAUSE, or GENERAL-NP. If you add a kernel specification, you will be asked to select the realization function for the kernel: a CURRIED-REALIZATION-CLASS, a REALIZATION-CLASS, or a SINGLE-CHOICE.

After adding a new specification, the message will redisplay with placeholders for the pieces of the new specification which you must add. For example, if you add a bundle specification, there will be placeholders for the bundle head, for required accessories which don't take values (if necessary), and for accessory values of required accessories (if necessary). The placeholders will be !! ADD BUNDLE HEAD !![2], !! ADD REQUIRED ACCESSORY !!, and !! ADD ACCESSORY VALUE !!, respectively. The placeholders for kernel specifications will be descriptive names for the arguments of the kernel. For example, if you chose the curried realization class FREE as the realization function of the kernel specification, then the placeholders for the arguments of the kernel specification would be AGENT and PATIENT.

You must specify values for all placeholders before running or saving the current message.

Bundle heads

To replace or add a bundle head, you will first select the type of bundle head: BUNDLE SPECIFICATION, KERNEL SPECIFICATION, or EXISTING SPECIFICATION.

[2] The exclamation points are part of the names. They signify that these operations are required. Operations that are optional are indicated with *'s, for example * ADD ACCESSORY *.

Then you will perform the necessary operations depending upon what type of specification you just chose.

It is possible to add accessories to a bundle specification. Each bundle type has a predefined list of associated accessories. You can add any of the associated accessories which have not already been specified.

Accessories

To add an accessory, place the mouse over the piece of the input message which says, * ADD ACCESSORY * and click any mouse button. Then select the ADD option from the Options menu and a menu will come up which lists the accessories you can add to the bundle associated with this * ADD ACCESSORY * option. Selecting one of the accessories adds that accessory to the bundle. If the accessory takes a value, you will be asked to select a value for that accessory. For example, if you decide to add the GENDER accessory, you must select MASCULINE, FEMININE, or NEUTER as the value of that accessory, and if you add the WH accessory, you must give a bundle or kernel specification as the value.

The following list shows all the accessory types and their possible values, if any.

ACCESSORY TYPE	ACCESSORY VALUE
unmarked	no accessory value
question	no accessory value
perfect	no accessory value
progressive	no accessory value
command	no accessory value
negate	no accessory value
number	singular or plural
person	first, second, or third
gender	masculine, feminine, or neuter
determiner-policy	indefinite-first-mention_definite-subsequent-mentions, always-definite, no-determiner,
tense-modal	past, present, or any modal verb
wh	a bundle or kernel specification
given	a bundle or kernel specification
purpose-clause-object	a bundle or kernel specification

Required Accessories

Adding a required accessory to a bundle is slightly different from adding an accessory to a bundle. This option forces you to add one of the required accessories to a bundle specification when you have created a new bundle. (If you didn't add one of the required accessories to a bundle and you ran Mumble with the current message, you would get an error.)

To add a required accessory, place the mouse over the piece of the input message which says, !! ADD REQUIRED ACCESSORY !! and click any mouse button. Then select the Add option from the Options menu and a menu will come up which lists

the required accessories for the bundle which is associated with the *ADD REQUIRED ACCESSORY * option. You only need to select one of the required accessories. After selecting one of the required accessories, you will be asked to supply an accessory value if the accessory you just added takes a value.

Accessory Values

To replace or add an accessory value, you will select the new accessory value from a list of possible values (or you might create a specification for those accessories whose value is a specification). See above for the possible values for each accessory type.

Further Specifications

To add a further specification to a bundle, place the mouse over the piece of the input message which says, * ADD FURTHER SPECIFICATION * and click any mouse button. Then select the ADD option from the Options menu. When the message is redisplayed, there will be placeholders for the attachment function, !! ADD ATTACHMENT FUNCTION !!, and the specification, !! ADD SPECIFICATION !!. You must add these pieces to the further specification before running or saving the current message.

Attachment Functions

To replace or add an attachment function, you will first select the type of the new attachment function: SPLICING ATTACHMENT POINT, LOWERING ATTACHMENT POINT, or ATTACHMENT CLASS. Then you will select the actual attachment function.

Specification to a Further Specification

To replace or add a specification to a further specification, you will first select the type of specification: BUNDLE SPECIFICATION, KERNEL SPECIFICATION, or EXISTING SPECIFICATION. Then you will perform the necessary operations as described above depending upon what type of specification you just chose.

Realization Function of a Kernel Specification

To replace a realization function of a kernel specification, you must select the type of realization function: REALIZATION CLASS, CURRIED REALIZATION CLASS, or SINGLE CHOICE. Then you will select the actual realization function.

Arguments to a Kernel Specification

To replace or add an argument to a kernel specification, you must first select the type of argument: WORD, PRONOUN, BUNDLE SPECIFICATION, KERNEL SPECIFICATION, or EXISTING SPECIFICATION. If you select WORD or PRONOUN as the type, you will then select the argument from a list of those words or pronouns which have been defined. If you select BUNDLE SPECIFICATION, KERNEL SPECIFICATION, or EXISTING SPECIFICATION, you will perform the necessary operations as described above depending upon what type of specification you just chose.

7.3. SAMPLE EDITING SESSION

In this section we walk step by step through an editing session. To get the most out of the section, we suggest that you read it in the company of a lisp machine with Mumble-86 loaded and follow along through each step. (Note: The session was run on a Symbolics 3670 lisp machine running Genera 7.1.)

To start the Interactive Demo, click on the command INTERACTIVE DEMO from the Mumble Command Menu in any of Mumble's screen configurations.

Step 1: To begin editing a message, select the Edit Message command from the Interactive Demo Command Menu which is in the upper right-hand corner of the screen. After selecting this command, you will see the list of Mumble input messages which have been defined and which you can edit.

Click on the message named FLUFFY-THE-LITTLE-DOG and this message will be printed in the Message Window as shown in Figure 7.2.

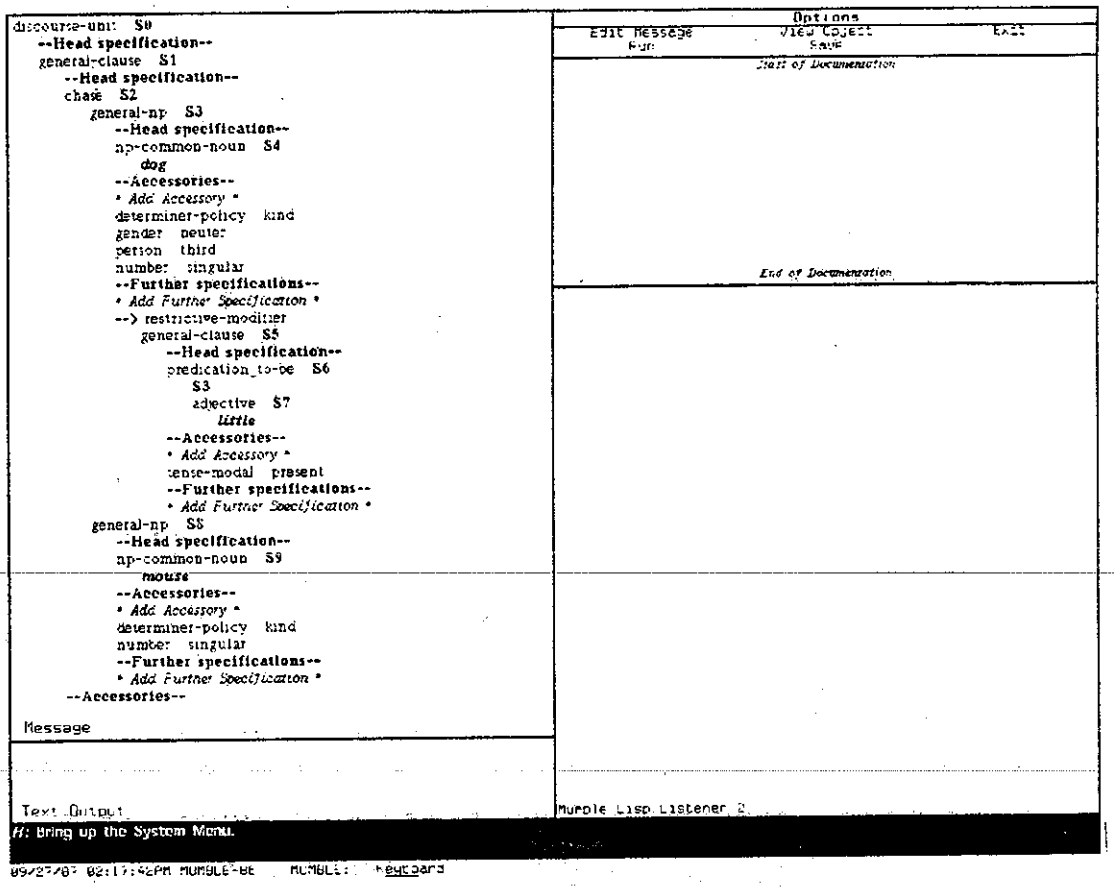
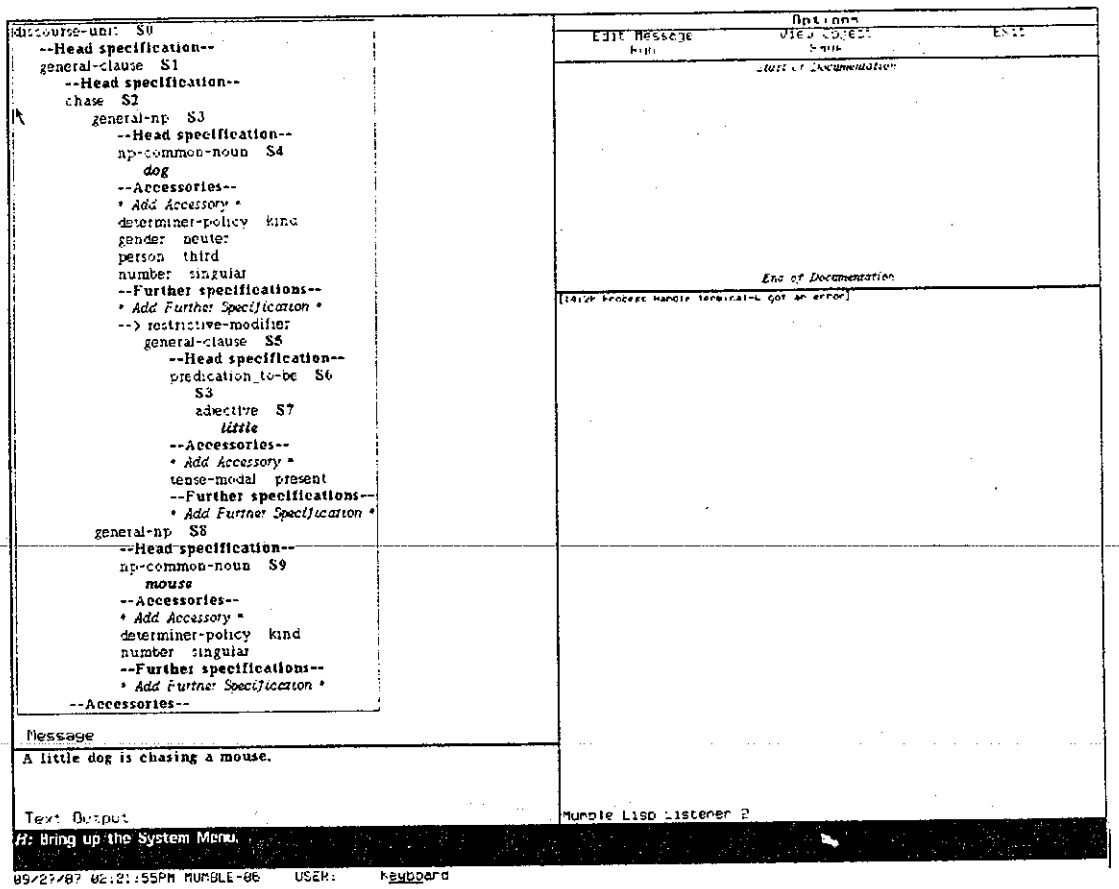


FIGURE 7.2 MESSAGE NAMED FLUFFY PRINTED IN THE MESSAGE WINDOW

Step 2: Try running Mumble with the current message in the Message Window by selecting RUN from the Command Menu. The text Mumble produces will appear in the Text Output Window in the lower left-hand corner of the screen. The output for the current message is the sentence, *A little dog is chasing a mouse.*

Step 3: By moving the mouse over the text in the Message Window, it is possible to outline particular pieces of the input message. When a piece of the input message is outlined, the documentation line at the bottom of the screen gives a short descriptor for that piece. Move the mouse so the entire message is outlined as in Figure 7.3. The entire message is a discourse unit and is labeled DISCOURSE-UNIT S0. The documentation line gives the following descriptor for DISCOURSE-UNIT S0: #<BUNDLE-SPECIFICATION DISCOURSE-UNIT for GENERAL-CLAUSE for CHASE>.



7.3 ENTIRE MESSAGE, DISCOURSE-UNIT S0

Before editing any part of the message, try moving the mouse around and outlining different pieces of the message. For example, DISCOURSE-UNIT S0 has a head specification named GENERAL-CLAUSE S1. GENERAL-CLAUSE S1 has a head specification named CHASE S2 and three accessories, UNMARKED, PROGRESSIVE, and TENSE-MODAL. All the accessories may not be on the screen, but the Message Window is scrollable (see Section 7.1).

The kernel specification CHASE S2 is shown outlined in Figure 7.4.

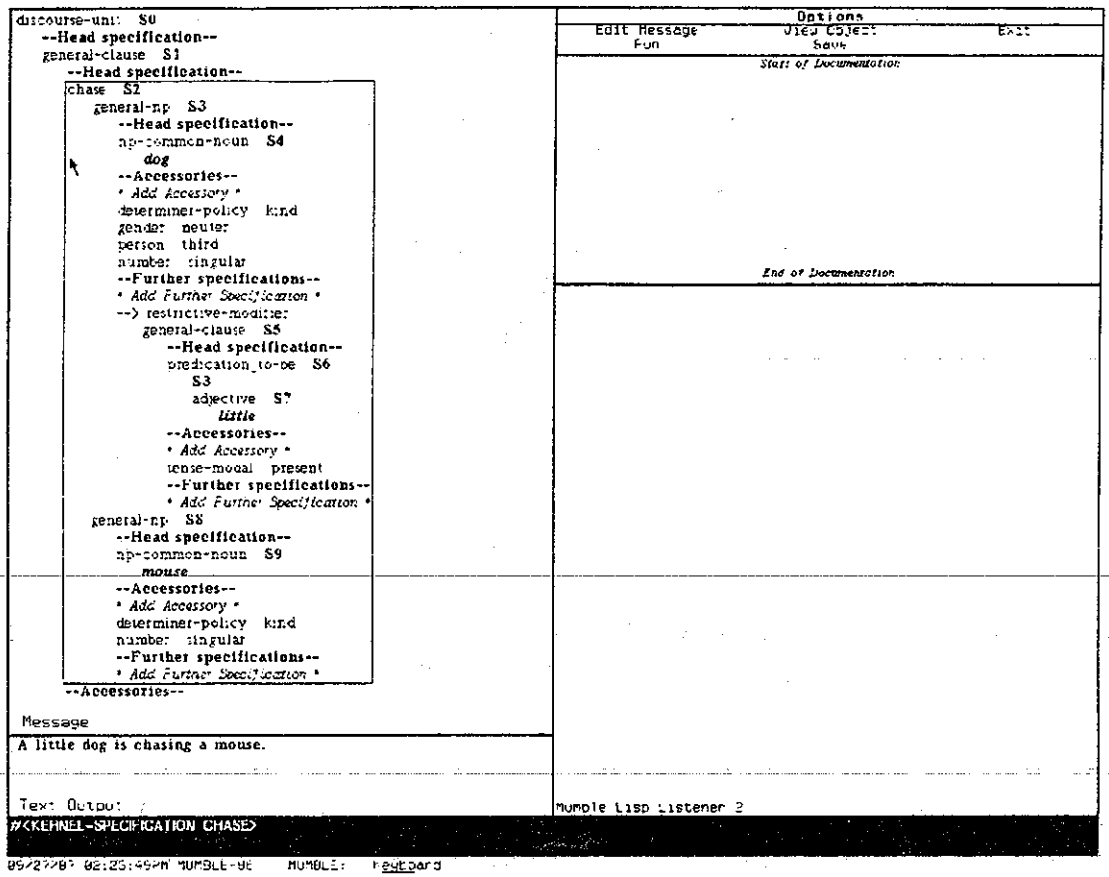


FIGURE 7.4 CHASE S2

This specification has two arguments: GENERAL-NP S3 and GENERAL-NP S8. GENERAL-NP S3 has a head specification named NP-COMMON-NOUN S4, four accessories, and one further specification. GENERAL-NP S8 has a head specification named NP-COMMON-NOUN S9 and two accessories.

Step 4: To get the Options menu for any piece of the input message, place the mouse so a piece of the message is outlined and then click any mouse button. Outline the specification, CHASE S2, as shown in Figure 7.4 and then click any mouse button. The Options menu will come up and it lists three options: REPLACE, VIEW OBJECT, and HELP. Notice that since CHASE S2 is the head of the bundle specification, GENERAL-CLAUSE S1, the Options menu which comes up is for bundle heads.

Step 5: To display help information, select the HELP option from any Options menu. Get help information for the specification, CHASE S2. Outline CHASE S2,

click any mouse button to get the Options menu, and then select HELP from the Options menu. Help information will be displayed in the Documentation Window.

Step 6: To view any Mumble object, either select the VIEW OBJECT option from any Options menu, or select the VIEW OBJECT command from the Interactive Demo Command Menu. If selecting VIEW OBJECT from an Options menu, the piece of the message which was currently outlined will be described in the Lisp Listener. If selecting VIEW OBJECT from the Command Menu, you will first select the type of object to view, then the particular object to view, and then that object will be described in the Lisp Listener.

First, view the specification CHASE S2. Get the Options menu for CHASE S2 and select VIEW OBJECT from the menu. The specification will be displayed in the Lisp Listener.

Now view the curried realization class, CHASE. Select VIEW OBJECT from the Command Menu and then select CURRIED-REALIZATION-CLASS from the next menu which comes up. Finally, select CURRIED-REALIZATION-CLASS CHASE from the next menu and it will be displayed in the Lisp Listener.

Step 7: You are now ready to edit the current message. Begin by replacing an accessory value for GENERAL-NP S3, the first argument of the kernel specification CHASE S2. Place the mouse in the Message Window so GENERAL-NP S3 is outlined. Remember that GENERAL-NP S3 has a head specification (NP-COMMON-NOUN S4), four accessories (DETERMINER-POLICY, GENDER, PERSON, and NUMBER), and one further specification which is a RESTRICTIVE-MODIFIER. Replace the value of the NUMBER accessory. Its current value is SINGULAR. To replace the accessory value, place the mouse over SINGULAR, click any mouse button to get the Options menu, and then select REPLACE from the Options menu. A menu will come up which lists two objects: ACCESSORY-VALUE SINGULAR, and ACCESSORY-VALUE PLURAL. Select ACCESSORY-VALUE PLURAL, so PLURAL will replace SINGULAR as the accessory value for the NUMBER accessory. The message in the Message Window will be redisplayed to reflect this change. Wait for the message to be redisplayed and then run Mumble with the new message (by selecting RUN in the Command Menu). Mumble's output is now, *Little dogs are chasing a mouse* . The subject of the sentence, *dog*, is now plural.

Now replace the accessory value for the TENSE-MODAL accessory in GENERAL-CLAUSE S1. The current value of the TENSE-MODAL accessory is PRESENT. You may have to scroll the Message Window in order to replace this accessory value. Click on PRESENT to get the Options menu and then select REPLACE from the menu. Another menu will come up which lists all the possible values for the TENSE-MODAL accessory. Select WORD COULD from the list of possible values. Wait for the

message to redisplay and then run Mumble. Mumble's output is now, *Little dogs could be chasing a mouse.*

Step 8: Add an accessory to GENERAL-CLAUSE S1. Click on * ADD ACCESSORY * as shown in Figure 7.5 and an Options menu will come up. (Notice that the message in the Message Window has been scrolled in order to add an accessory).

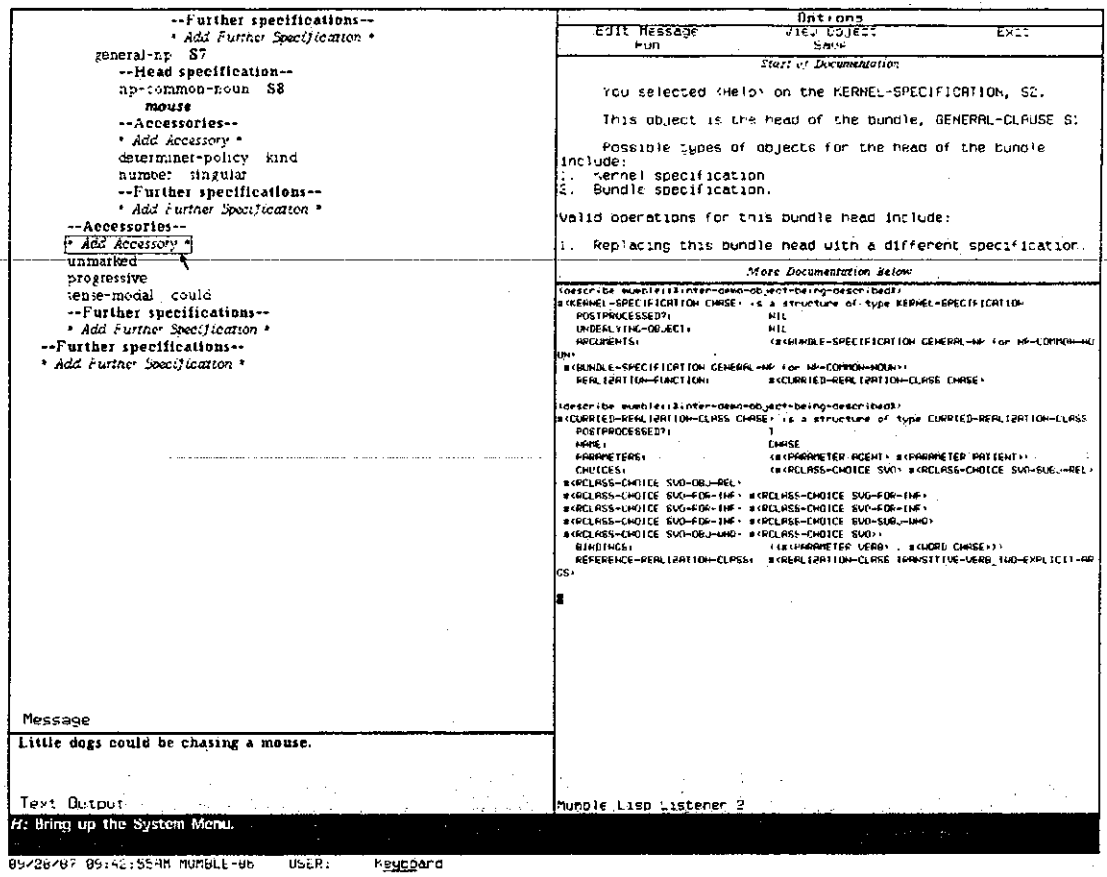


FIGURE 7.5 THE * ADD ACCESSORY * OPTION FOR GENERAL-CLAUSE S1

Select ADD from the menu and another menu comes up which lists all the accessories which can be added to the bundle GENERAL-CLAUSE S1. Add the accessory type PERFECT by selecting ACCESSORY-TYPE PERFECT from the menu. Wait for the message to redisplay and then run Mumble. The output from Mumble is now - *Little dogs could have been chasing a mouse.*

Step 9: Delete the accessory you just added to the message. To do this, click on PERFECT and select DELETE from the Options menu. Wait for the message to

redisplay and then run Mumble. The output is now - *Little dogs could be chasing a mouse.*

Step 10: Delete the further specification from GENERAL-NP S3. Place the mouse over the further specification as in Figure 7.6.

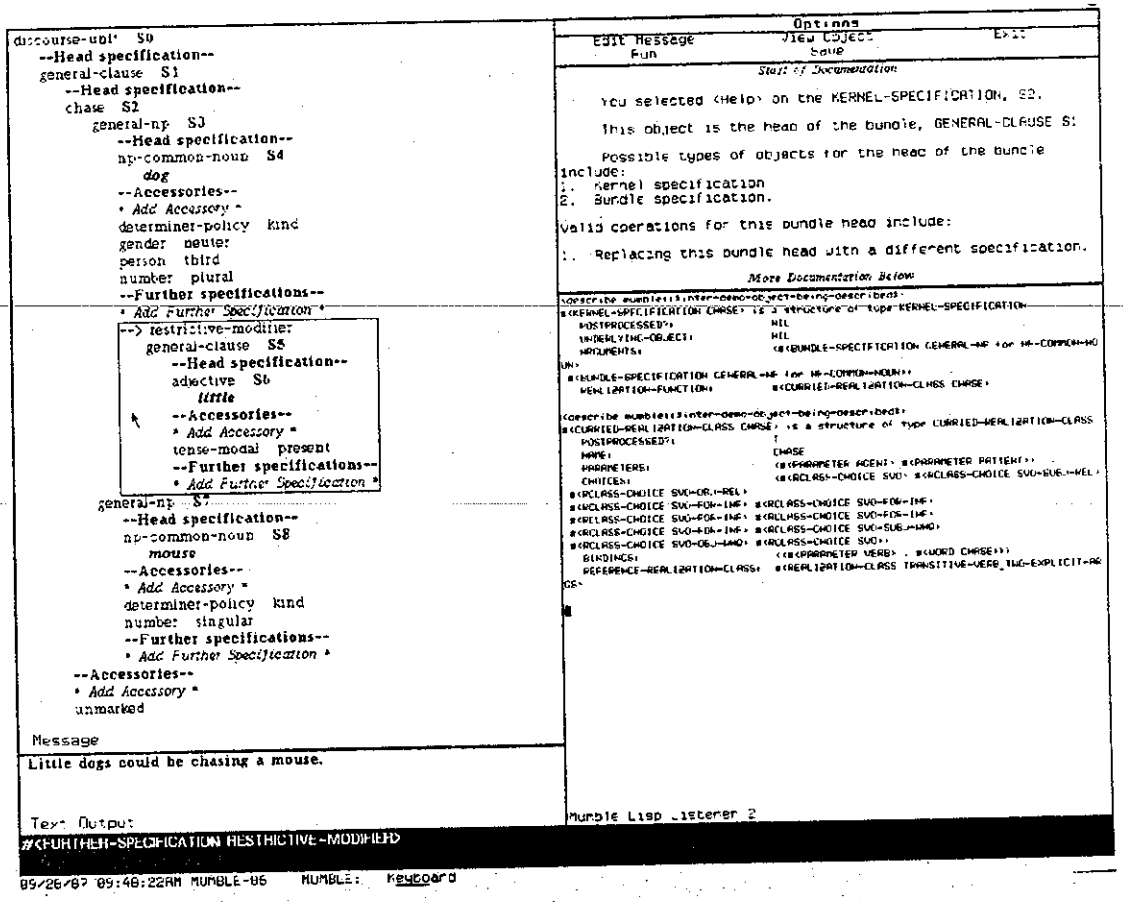


FIGURE 7.6 FURTHER SPECIFICATION, RESTRICTIVE MODIFIER

Click on any mouse button to get the Options menu and then select Delete from the Options menu. Wait for the message to redisplay and then run Mumble. Mumble's output is now *Dogs could be chasing a mouse.*

Step 11: Replace the word DOG which is an argument to the kernel specification NP-COMMON-NOUN S4. Get the Options menu for DOG and select REPLACE from the menu. Another menu will comes up which lists the types of kernel specification arguments: WORD, PRONOUN, BUNDLE SPECIFICATION, KERNEL SPECIFICATION, and EXISTING SPECIFICATION.

Select WORD from the menu in order to replace the word DOG with another word[3]. After selecting WORD from the menu, another menu will come up which lists all the types of words which have been defined.

Replace DOG, which is a noun, with another noun by selecting NOUN from the menu. Then a menu will come up which lists all the nouns which have been defined. Select CAT from the list of nouns and wait for the message in the Message Window to redisplay. Now run Mumble with the new message and the output is *Cats could be chasing a mouse.*

Step 12: Replace the realization function for the kernel specification CHASE S2. Place the mouse over CHASE as in Figure 7.x and click the mouse to get the OPTIONS menu.

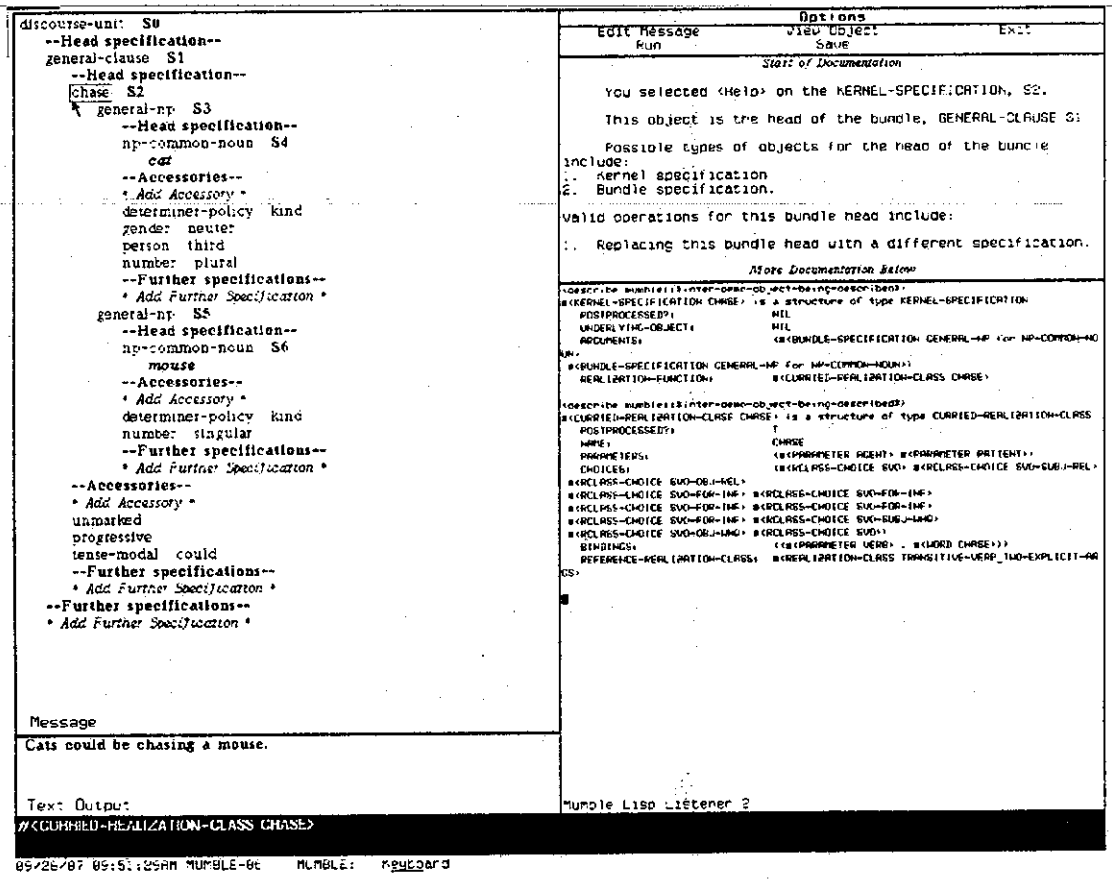


FIGURE 7.7 REALIZATION FUNCTION FOR CHASE S2

[3] It is always a good idea to replace one type of object with an object of the same type. Using an object of a different type as a replacement may not always produce a message which will run when given to Mumble.

Select REPLACE from the Options menu and a menu will be displayed which contains the types of objects which could be realization functions for the kernel specification. The three possible types are: REALIZATION CLASS, CURRIED REALIZATION CLASS, and SINGLE CHOICE. Select CURRIED REALIZATION CLASS from the menu and another menu will come up which lists all the curried realization classes which have been defined. Select CURRIED-REALIZATION-CLASS BITE from the list and BITE will replace CHASE as the realization function for CHASE S2. Wait for the message to redisplay and then run Mumble. The new output is, *Cats could be biting a mouse*. Note also that the name of the kernel specification is now BITE S2, instead of CHASE S2.

Step 13: The rest of this editing session will show how to add new specifications and how to add all parts of the specification. We will start by replacing GENERAL-CLAUSE S1 with a new bundle specification. (GENERAL-CLAUSE S1 is the head of the bundle specification, DISCOURSE-UNIT S0.) Outline GENERAL-CLAUSE S1, click any button to get the Options menu, and select REPLACE from the menu which comes up. The next menu which appears will ask for the type of specification with which to replace the current bundle head. Select BUNDLE SPECIFICATION from the menu. The next menu which comes up will prompt you for the type of the bundle to add. Select GENERAL-CLAUSE from the menu and then wait for the message to redisplay.

Just for fun, run Mumble with the current message. Notice that you are not able to run Mumble because some essential parts of the message have not been specified, namely the bundle head and a required accessory.

Step 14: Add one of the required accessories to the new bundle specification, GENERAL-CLAUSE S1. Get the Options menu by outlining !! ADD REQUIRED ACCESSORY !! and then clicking any mouse button. Select ADD from the menu. Then a menu will come up which lists the required accessories for this bundle. You only need to add one of these accessories to the bundle. Select UNMARKED from the required accessory list and wait for the message to redisplay.

Step 15: In order to specify the tense of the clause we are building, add the TENSE-MODAL accessory to GENERAL-CLAUSE S1. First get the Options menu for adding an accessory by clicking on the * ADD ACCESSORY * option for GENERAL-CLAUSE S1. Then select ADD from the Options menu. Select the TENSE-MODAL accessory from the next menu which comes up and then select WORD CAN from the menu which prompts you for the value of the tense-modal accessory. Wait for the message to redisplay.

Step 16: Add a bundle head to the new GENERAL-CLAUSE S1. First get the Options menu by outlining !! ADD BUNDLE HEAD !! and then clicking any mouse button. Select ADD from the Options menu. Then select KERNEL SPECIFICATION from the next menu which comes up to make the bundle head a kernel specification. The next menu which comes up will be prompting you for the realization function of the kernel specification which you are adding. Select CURRIED-REALIZATION-CLASS from the menu and then select the curried realization class READ from the next menu which comes up. Wait for the message to redisplay and notice that a kernel specification has been added to the message - its name is READ S2 and it has two arguments: AGENT and PATIENT. These two arguments must be added to the message before running or saving the current message.

Step 17: Add the arguments, AGENT and PATIENT, to the kernel specification READ S2. First add the argument, AGENT. Outline AGENT, get the Options menu, and select ADD from the menu. Select BUNDLE-SPECIFICATION from the next menu in order to make this argument a bundle specification. Then select GENERAL-NP as the type of this bundle. Wait for the message to redisplay. The pieces of this bundle which you must specify are the bundle head and the values of the four accessory types: PERSON, NUMBER, GENDER, and DETERMINER-POLICY. First add the bundle head. Get the Options menu for the bundle head, select ADD from the menu, and make this bundle head a kernel specification. Select SINGLE-CHOICE as the type of realization function for the kernel specification and then select the single choice, NP-PROPER-NAME, as the realization function. Wait for the message to redisplay. The new kernel specification is named NP-PROPER-NAME S4 and its only argument is NOUN. Add this argument now. Get the Options menu for NOUN, select ADD from the menu, select WORD as the type of argument, select PROPER-NOUN as the type of word, and then select PETER as the proper-noun. Wait for the message to redisplay.

Now add the second argument, PATIENT, to the kernel specification READ S2. Get the Options menu for PATIENT, select ADD from the menu, select BUNDLE-specification from the next menu, and then select GENERAL-NP as the type of bundle specification. The new bundle is named GENERAL-NP S5. Notice again that you must add a bundle head and four accessory values to a bundle of type general-np. Add the bundle head first. Get the Options menu for adding the bundle head, select ADD from the menu, select KERNEL SPECIFICATION as the type of bundle head, select SINGLE-CHOICE as the type of kernel specification realization function, and finally select NP-COMMON-NOUN as the realization function. Wait for the message to redisplay. The new kernel specification is named NP-COMMON-NOUN S6 and it only has one argument, NOUN. Add this argument now. Get the Options menu for NOUN and select ADD from the Options menu. Then select WORD as the type of

argument, select NOUN as the type of word, and then select BOOK as the noun. Wait for the message to redisplay.

Step 18: Now the only pieces of the message which still need to be added are accessory values for the bundles, GENERAL-NP S3 and GENERAL-NP S5. Four accessory values must be specified for each bundle. First add the accessory values for the bundle GENERAL-NP S3. Get the Options menu for the accessory value for PERSON by outlining the !! ADD ACCESSORY VALUE !! which is opposite PERSON. Select ADD from the Options menu and then select THIRD as the accessory value. Wait for the message to redisplay and add accessory values for the other three accessories. Add the values as follows: number should be SINGULAR, GENDER should be MASCULINE, and DETERMINER-POLICY should be NO-DETERMINER. After those three values have been added, add the four accessory values for the bundle, GENERAL-NP S5. The accessory values should be as follows: PERSON should be THIRD, NUMBER should be PLURAL, GENDER should be NEUTER, and DETERMINER-POLICY should be KIND. Now you can run Mumble with the message, because all required parts of the message have been specified. The output is now *Peter can read books.*

Step 19: Add a further specification to GENERAL-NP S5. Get the Options menu for adding a further specification by outlining the * ADD FURTHER SPECIFICATION * option for GENERAL-NP S5. Select ADD from the Options menu and wait for the message to be redisplayed. Notice that there are two placeholders for the two parts of the further specification. These placeholders are !! ADD ATTACHMENT FUNCTION !! and !! ADD SPECIFICATION !!. You must add an attachment function and a specification to the further specification.

Step 20: Add an attachment function to the further specification you just created. Get the Options menu for adding an attachment function by outlining !! ADD ATTACHMENT FUNCTION !! and clicking any mouse button. Select ADD from the Options menu. The next menu which comes up will be prompting you for the TYPE of attachment function. Select SPLICING ATTACHMENT POINT. Then select ADJECTIVE from the next menu which appears; the actual attachment function you want to use is ADJECTIVE. Wait for the message to redisplay.

Step 21: Add a specification to the further specification. Get the Options menu for adding a specification and then select ADD from the menu. Select KERNEL SPECIFICATION as the type of new specification and then select SINGLE CHOICE as the realization function for the kernel specification. When asked for the single choice, select ADJECTIVE. Wait for the message to redisplay. The new message will

contain a placeholder for the one argument to the kernel specification you just added. This argument is ADJECTIVE. Add this argument now. Get the Options menu for ADJECTIVE, select ADD from the menu and then select WORD as the type of argument you want to add. The next menu which comes up will want you to select the TYPE of word; select ADJECTIVE. When a menu of adjectives comes up, select the adjective, BIG. Wait for the message to redisplay and then run Mumble. The output is *Peter can read big books.*

Step 22: Save the message you have just created. To do this, select SAVE from the Command Menu. Wait for the message to come up which tells you the message has been saved. Then look at the message in the Zmacs buffer named SAVED-MESSAGES. Before doing anything in the buffer, reparse the attribute list for the buffer. You can do this by hitting META-X and then typing REPARSE ATTRIBUTE LIST. Pretty-print the message in the buffer by placing the mouse on the left parentheses before DEF-DEMO-RSPEC and then typing the META-CTRL-Q keystroke. The SAVED-MESSAGES buffer should look similar to Figure 7.8.

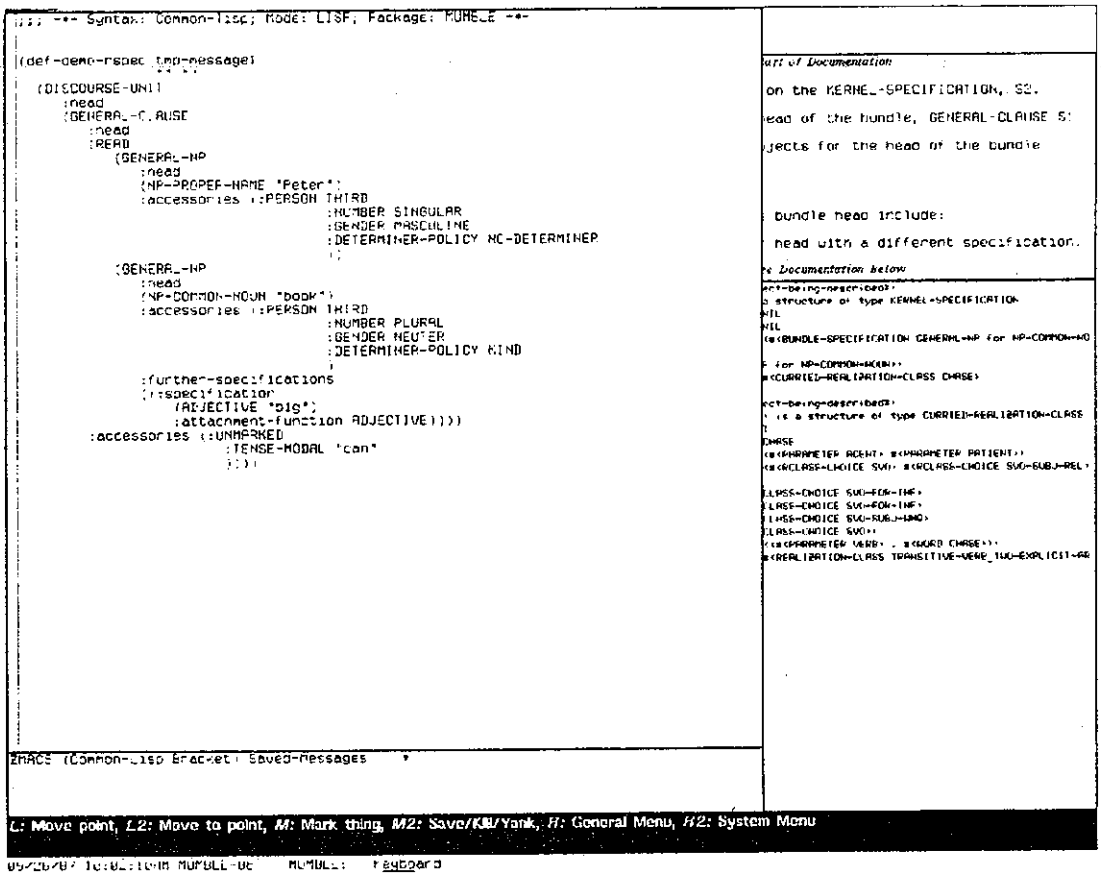


FIGURE 7.8 SAVED-MESSAGES

Evaluate the message. Notice that the name of the message is TMP-MESSAGE1. Go back to the Mumble screen and select the EDIT MESSAGE command. Notice that TMP-MESSAGE1 is now on the list of demos which you can edit.

Getting out of an Error

You are now ready to edit on your own. If you get an error while running Mumble, hitting the ABORT key is usually the safest way to proceed from the error. When you are finished using theInteractive Demo, select EXIT from the Command Menu and you will bereturned to the Mumble Configuration from which the Interactive Demo was called.

8. REFERENCES

Conklin, E. Jeffery (1983) Data-Driven Indelible Planning of Discourse Generation Using Salience, Ph.D. Dissertation, Department of Computer & Information Science, University of Massachusetts at Amherst, June 1983; available as Technical Report 83-13.

Conklin, E. Jeffery, Kate Erlich, & David McDonald (1983) "An Empirical Investigation of Visual Salience and Its Role in Text Generation", **Cognition and Brain Theory**, 6(2) Spring 1983, pp. 197-225.

Joshi, Aravind (1985) "How much context-sensitivity is required to provide reasonable structural descriptions: tree adjoining grammars", in Dowty, Karttunen & Zwicky (eds.) **Natural Language Processing: Psycholinguistic, Computational and Theoretical Perspectives**, Cambridge University Press.

Joshi (1987) "The Relevance of Tree Adjoining Grammar to Generation", in Kempen (ed.) **Natural Language Generation**, Martinus Nijhoff, Dordrecht, The Netherlands.

McDonald, David D. (1974) "Conversations Between Programs", MIT Artificial Intelligence Lab working paper 76, May 1974.

McDonald (1975) "A Preliminary Report on a Program for Generating Natural Language", **IJCAI-75**, Tibilisi USSR, August, 1975, pp. 401-405.

McDonald (1978a) "A Simultaniously Procedural and Declarative Data Structure and its Use in Natural Language Generation", proc. **CSCSI**, Univ. of Toronto, July 5-7, 1978, pp. 38-47.

McDonald (1978b) "Language Generation: Automatic Control of Grammatical Detail", **COLING-78**, Bergen Norway, August, 1978, pp. 52-55.

McDonald (1980) **Natural Language Production as a Process of Decision-making under Constraints**, unpublished Ph.D. Dissertation, MIT Artificial Intelligence Laboratory, August 1980.

McDonald (1981) "Language Production: the Source of the Dictionary", **ACL**, Stanford University, June 29 - July 1, 1981, pp. 57-62.

McDonald (1983) "Natural Language Generation as a Computational Problem: an introduction", in Brady & Berwick (eds.) **Computational Models of Discourse**, MIT Press, 1983, pp. 209-266.

McDonald (1984) "Description Directed Control: Its implications for natural language generation", **International Journal of Computers and Mathematics**, 9(1) Spring, 1983, pp. 403-424; whole issue reprinted as **Computational Linguistics**, Cercone (ed.), Plenum Press, 1984; article later reprinted in Grosz, Spark-Jones & Webber (eds.) **Readings in Natural Language Processing**, Morgan Kaufmann, 1986, pp. 519-537.

McDonald & Jeff Conklin (1982) "Salience as a Simplifying Metaphor for Natural Language Generation", **AAAI-82**, August 18-20, 1982, Carnegie-Mellon University, pp. 75-78.

McDonald & Marie Meteer (1987) "From Water to Wine: Generating Natural Language Text from Today's Applications Programs", TR #87-51, Department of Computer & Information Science, University of Massachusetts at Amherst.

McDonald, Meteer & James Pustejovsky (1987) "Factors contributing to efficiency in Natural Language Generation", in Kempen (ed.) **Natural Language Generation**, Martinus Nijhoff, Dordrecht, The Netherlands.

McDonald & James Pustejovsky (1985a) "A Computational Theory of Prose Style for Natural Language Generation", **ACL** (European chap.), Univ. of Geneva, March 28-30, 1985, pp. 86-94.

McDonald & Pustejovsky (1985b) "TAG's as a Grammatical Formalism for Generation", **ACL**, Univ. of Chicago, July 8-12 1985, pp. 94-103.

McDonald & Pustejovsky (1985c) "Description-Directed Natural Language Generation", **IJCAI-85**, UCLA, August, 18-23, 1985, pp. 799-805.

Rubinoff, Robert (1986) "Adapting Mumble: Experience with Natural Language Generation", proc. AAAI-86, Philadelphia PA, August 11-15, 1986, pp.1063-1068, Morgan-Kaufman.