The production of prosodic focus and contour in dialogue

Thesis

How to cite:

For guidance on citations see FAQs.

© 1992 The Author

Version: Version of Record

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online’s data policy on reuse of materials please consult the policies page.
Chapter 5

Prosody and language production in Sundial

5.1 Overview of the Sundial dialogue manager

This chapter presents details of an implementation which instantiates the models proposed in Chapters 3 and 4. The Sundial system is a prototype dialogue system, under development as part of a collaborative project aimed at providing telephone interfaces to a database (Peckham 1991). Figure 5.1 shows an architectural overview. Amongst the goals of the project which affect this work are those of achieving generality and language-independence in the knowledge bases, and naturalness of language and dialogue as observed in studies of human-human and human-machine conversations.

The system shown in Figure 5.1 has three major components: the Voice Input and Message Output subsystems, and the Dialogue Manager. Only the latter two components are relevant to this thesis. The system as a whole is a result of joint work. The break-down of responsibility for the individual components was explained in Section 1.5. This section gives an overview of the architectural and functional aspects of the Sundial dialogue manager as a whole. This is related to the computational model proposed in Chapter 3.

Thereafter I concentrate on those parts of the system which are my own work. Section 5.2 describes the implementation of the Belief Module, which corresponds
closely to the discourse interpretation component of Chapter 3. Section 5.3 relates to the planning part of message generation, that is, deriving a graph structure from within the discourse model which corresponds to the required message, together with information relating to the relative prosodic prominence of the semantic constituents. In Section 5.4 I describe the linguistic part of message generation, whereby surface structure is produced from the conceptual input, using a lexical knowledge base and a linguistic history. Section 5.5 demonstrates how prosodic markings at the surface level, including the choice of prosodic contours, are derived. I also give an overview of the mechanisms whereby the higher-level prosodic decisions are eventually acoustically realised. Finally, In Section 5.7 I give a detailed example which brings together the workings of these parts.

5.1.1 Architecture

Viewed from the outside, this component of the voice transaction system has the function of making sense of voice input to which linguistic analysis has already been
applied, and producing its own messages for generation in the voice output component. Internally, it has to manage transactions with an application component on the Caller's behalf, reason about its own contributions and the Caller's, ensuring that the conversation unfolds in an appropriate manner, and maintain contextual representations that will facilitate the interpretation of future input and the generation of future messages. It has also to provide some counterweight to the inadequacies of current speech input technology: both by deriving predictions about what the Caller's next contribution might be and by being prepared to engage in repair actions when communication failure occurs.

Architectural generality and portability have been achieved by separating out knowledge and control among a number of specialised agents, or modules. These modules are shown as part of the overall dialogue manager, in Figure 5.1. Central among these is the Dialogue Module. This is responsible for decisions concerning what to say next, and the pragmatic interpretation of input. Communication between task-level goals and responses and the outside world is managed, with the requirement that conversational coherence be adhered to. The dialogue module also manages repair situations, openings and closings. It does all this by virtue of its specialised knowledge about dialogue. A set of dialogue rules specify what well-formed dialogue structure should look like. The model follows closely that of Moeschler (1986), discussed already in Section 3.2.1. That is, dialogue moves are organised hierarchically, according to what is effectively a dialogue grammar. At the lowest level are dialogue acts; the term intervention covers both these and compound dialogue acts. At the next level up, exchanges consist of slots labelled as initiatives, reactions and evaluations. These may be filled either by interventions, or by embedded exchanges. At the uppermost level is the transaction, which consists of a number of exchanges. The interaction is advanced by maintaining a complete structured record of the preceding dialogue, and attempting to extend this dialogue history either as a result of interpreting an input from the Caller, or by the system producing a message of its own. In addition to the dialogue rules, which specify well-formed dialogue in the abstract, the module uses a set of precondition-effect rules corresponding to system dialogue acts. At every cycle, possible extensions to
the dialogue history are generated; those that correspond to the system speaking next are matched against the preconditions, and if further goals in the environment are met and the conflict set can be adequately reduced, the corresponding messages are sent for generation and the history updated accordingly. These goals are then satisfied; examples are requests for task parameters, or the need to confirm some input when the signal was doubtful. While system output is being generated, predictions of Caller responses can also be made. The interpretation of input needs to match upcoming expectations in the dialogue history. Also used are knowledge about fixed forms and discourse markers, and the result of interpretation in the Belief Module.

The Task Module is the component which manages interaction with the application. In the implementation described here, the application is modelled using a Prolog database. Guarding this are a set of rules which act as meta-constraints. These are used to ensure that a task is adequately specified, before a database access can be made. Until that time, the Task Module directs interaction to find out more information from the Caller.

The Linguistic Interface module sends out predictions to the linguistic analysis components, and reads in an analysis tree. This has nodes structured like signs, with syntactic and semantic components. The tree is stored in the linguistic history; its semantic component is sent on for further interpretation, and as a result unbound variables denoting unspecified nodes in a semantic network are given bindings. The Belief Module maintains the knowledge base that constitutes discourse memory, which other modules can access and update. It manages semantic interpretation with respect to this knowledge base, updating it where necessary, and drawing necessary inferences. It is discussed in more detail in Section 5.2.

The message planner takes instructions to generate a message, which may consist of more than one dialogue act, and produces a plan in the form of a semantic structure together with other necessary features, that can be used to drive the linguistic generator. In so doing it consults the Belief Module knowledge base, and updates it appropriately. Message planning is discussed further in Section 5.3.

Communication between modules is by message passing. A message can be a
request for information, or an instruction to a module to update its current internal representations in some way. It must be explicitly addressed to a destination module. Modules become active when a message is passed to them; in the current implementation, only one module at a time is active. The scheduling of messages, and therefore of processes, is handled by a supervisory module known as ‘postie’ (postmaster). Messages are kept on a first-in first-out queue; modules are activated when they are the next recipients in line.

5.1.2 The interpretation and generation cycles

Consider a stage in processing where the structure of the dialogue history could be advanced one step by the addition of input from the user. The Dialogue Module computes the possible options for this input, at a non-specific level, and sends these to the linguistic interface as predictions. Subsequently the linguistic interface returns a semantic expression representing the user’s utterance. Discourse interpretation is carried out by the belief module, which first strips off any features of pragmatic significance. The Dialogue Module then uses the results of discourse interpretation and rules relating pragmatic features to dialogue acts, to arrive at a pragmatic interpretation of the utterance. Any actions attached to the dialogue act rule that has been used are carried out, and the act added to the dialogue history. Suppose one such action is to inform the Task Module about the input. A message is sent to this module, which assumes control. If as a result of the recent input, the current task is sufficiently specified according to the application-specific meta-constraints, the task is forwarded to the application to be executed; otherwise the meta-constraints will result in further information being requested from the user.

In either case, the goal of producing a message is added to the Dialogue Module’s current goals. The generation cycle begins with an attempt to extend the dialogue history, having discovered that a system message is needed. The system dialogue act rules are then examined, to find those with the potential of advancing the current state by performing pending goals, such as that of informing the Caller. Candidate rules must also be compatible with the current dialogue history. As a result, the
message planner is sent instructions to output a certain set of dialogue acts. Details of how this is done are discussed more fully in Sections 5.3 and 5.2.

5.1.3 The computational model and the implementation

In Chapter 3 I presented a specification for a computational model of a conversing agent. It is the purpose of this section to relate the Sundial implementation described in this chapter to that model. For ease of reference, I refer to these as SUNDIAL and AGENT. I compare the overall architectures, and show how the modules correspond, so that SUNDIAL and AGENT in general produce the same behaviour. Finally, I point to those features of AGENT which do not exist in the version of SUNDIAL I am describing here, and those features of SUNDIAL not covered in AGENT.

Architecturally, both SUNDIAL and AGENT are made up of a small number of communicating processes. Refer to Figure 3.2 for the architecture of AGENT. Table 5.1 shows the correspondence between modules. Not shown are the input components of SUNDIAL, which correspond to the component INPUT of Figure 3.2.

In both architectures, processes (or modules) communicate by sending messages to one another. In the SUNDIAL implementation, control passes sequentially between modules, according to a schedule dictated by the queue of pending messages. The specification for AGENT on the other hand makes use of the algebra of Communicating Sequential Processes; recursive use of the notation enables it to be extended to describing the internal working of processes. Processes in principle operate in parallel, synchronising on common messages.
The simplest correspondence is between the SUNDIAL Task Module and the process INFO. Both are driven by a specification of the user's task. Both may require interactive refinement of the specification; culminating possibly in a successful database transaction, after which some result may be delivered.

In SUNDIAL, meta-communication (confirmations and corrections), phatic communication (openings, closings and holds), and pragmatic interpretation are all part of the functionality of the Dialogue Module. The system initiates confirmations and corrections on the basis of status reports from the Belief Module; initiation of phatic utterances results from Task Module goals; interpretation is carried out by special procedures. In AGENT I have divided these functionalities: only P_INT needs to directly communicate with the response stack, which corresponds to the SUNDIAL dialogue history. The process META is designed to intercept messages—requiring interactive ratification before either sending them on, modifying them, or suppressing them. In SUNDIAL interpreted messages are not directed further than the Dialogue Module. This decides what subsequent action to take.

An example showing the working of SUNDIAL, together with the corresponding messages and state changes within AGENT, will clarify the relationship. Consider the dialogue fragment in (5.1):

\begin{verbatim}
A1  can I help you
C1  what time does BA123 arrive in Paris
(5.1) A2  you want to know the arrival time of BA123
C2  yes
A3  BA123 arrives . . .
\end{verbatim}

the events corresponding to messages between SUNDIAL Modules are shown in Table 5.2. I use the same pictorial technique as in Chapter 3 for representing communicating processes. Horizontal lines represent processes, with time progressing from left to right. Thick arrows represent state transitions of those processes. Vertical lines represent messages between processes. For ease of labelling, messages are numbered. Channels are identified by pairs of modules names: thus mpbm is the channel from MP to BM.

Figures 5.2–5.3 show these events in juxtaposition with the corresponding events in AGENT. The events in AGENT are identified in the figure by lower-case char-
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>tmdm</td>
<td>request_start_task</td>
</tr>
<tr>
<td>2</td>
<td>dmmp</td>
<td>generate</td>
</tr>
</tbody>
</table>
| 3 | mpnmg | ukr_realize  
  A1  
  C1 |
| 4 | lidm | input_utterance |
| 5 | dmbm | request_anchor |
| 6 | bmdm | return_anchor |
| 7 | dmtm | start_task |
| 8 | tmbm | alt_solutions |
| 9 | tmdm | reply_solutions |
| 10 | dmmp | generate |
| 11 | mpnm | request_description |
| 12 | bmmp | return_description |
| 13 | mpnmg | ukr_realize  
  A2  
  C2 |
| 14 | lidm | input_utterance |
| 15 | dmmp | generate |
| 16 | mpbm | request_description |
| 17 | bmmp | return_description |
| 18 | mpnmg | ukr_realize  
  A3 |

Table 5.2: Messages between SUNDIAL modules
Figure 5.2: Events in SUNDIAL

Figure 5.3: Events in AGENT
Table 5.3: Event labels for AGENT

acters, explained in Table 5.3. Figures 5.2 and 5.3 are structured similarly, though AGENT has one additional process, META, and lacks an explicit LI module. It will be seen that AGENT models SUNDIAL closely, with the exception of the scheduling of access to the Task Module (INFO). In SUNDIAL, this is attempted (message 7) before confirmation has been carried out; this is to avoid potential presupposition violation. AGENT has been simplified, so that confirmation always takes place first. Only subsequently (message g) is access to INFO allowed.

I conclude this section by considering the mismatches in functionality between AGENT and SUNDIAL. Firstly, in AGENT the dialogue history is replaced by the response stack. Such a stack-based model of interaction has been used in the past, for example by Carberry (1988) and Litman (1985). However, as Bilange (1991) points out, a full dialogue history is to be preferred, if there are cases in which previously closed exchanges can be shown to be explicitly returned to. No clear cases of this occur in the corpora I have examined. Secondly, a major component of the functionality of SUNDIAL is left out of the account of AGENT: the generation of top-down predictions for use in parsing. Although of great importance to a speech dialogue system, this is not relevant to an account which concentrates on the production of output. Finally, the component OUTPUT is more flexible than the Message Output modules of SUNDIAL. In particular, it is designed to buffer messages, so that utterance production takes place in a lazy fashion, or chunk them,
so that production of a message may be interleaved with interaction with the user. The possibility of specifying such behaviour was a major factor in the choice of the CSP notation.

5.1.4 The message output components

The components of AGENT corresponding to the Belief Module, and the Message Output modules have deliberately received only scant description. They are of central importance to this thesis, since they implement the model proposed in Chapter 4. Viewed as a stand-alone component, the message output sub-system is capable, given the appropriate knowledge bases and inputs, of generating utterances such as those of the Agent shown in Appendix A, together with appropriate patterns of prosodic prominence. The implementation of this sub-system forms the subject of the remainder of this chapter.

5.2 Discourse modelling and accessibility

This section describes the implementation of the belief module, which as well as maintaining the discourse model, carries out the functions of interpretation and description. I describe first the knowledge representation language SIL and how the Belief Module is initialised with respect to SIL class definitions. I then give details of the discourse interpretation process. Consideration of the description routines is deferred until Section 5.3. Finally, I describe the implementation of the accessibility history, and show how it can be used to provide contexts for interpretation.

5.2.1 Knowledge representation

The Semantic Interface Language (SIL) has been used to cover different applications and domains, from flight reservations to train enquiries. SIL is language-independent and is being used for instances of the dialogue manager developed to handle English, French, German and Italian speakers.

The language (introduced in Section 4.3.1) uses the familiar notation of feature
structures to describe directed graphs. Every node in a SIL graph is typed; every type is associated with a number of possible roles which a node of that type may have. The language is specified in a set of class definitions, which specify what the types are and what local configurations are permitted for a given type. An example of a definition clause is given in (5.2). SIL is inheritance-based: a subsumption hierarchy is defined over the types, in such a way that if TYPE$_a$ inherits from TYPE$_b$, it will also by default inherit its roles. Figure 5.4 shows a portion of this hierarchy. Arrows denote ISA links; where relevant, roles defined for a class are shown, together with value restrictions. In the representation used here, all nodes inherit from a single node, TOP. Discourse entities are defined to behave as instances of nodes in this class hierarchy.

The definition in (5.2) defines the type location as having roles thecountry, thecity, theairport, thestation and theterminal:
Figure 5.5: *SiL* classes for locations and times

\[
\text{sdef} (\text{location}, \\
\text{location}_\text{property}, \\
[\text{thecountry}, \text{thecity}, \text{theairport}, \text{thestation}, \text{theterminal}], \\
[\text{thecountry}: \text{type} == \text{country}, \text{thecity}: \text{type} == \text{city}, \\
\text{theairport}: \text{type} == \text{airport}, \text{thestation}: \text{type} == \text{station}, \\
\text{theterminal}: \text{type} == \text{terminal}], \\
[\text{}])
\]  

The type location inherits from location\_property. Type constraints that the roles must satisfy are also defined. The definition of a type may also include inference rules which then apply to any instance of that type. The portion of the class hierarchy relating to locations and times is shown in Figure 5.5.

*SiL* is used to describe two kinds of datastructure: input/output graphs, and the graphs representing the states of the discourse model. Both kinds of graph obey the node constraints defined in the class definitions. However, input/output graphs are confined to what can be built up compositionally from the lexical entries which make up an utterance. They are directed acyclic graphs (DAGs) and therefore have distinguished nodes as roots, and no cycles. Discourse model graphs on the other hand have no distinguished roots, and may have cycles. They are constrained moreover by the integrity constraints—whereas for input graphs no attempt is made to enforce inferential closure, or to rule out structures which violate integrity constraints, such as *I want to fly from Paris to Paris*. This follows from the monotonic nature of semantic structure building during parsing. Input/output graphs also contain features not permitted on discourse model graphs (cf. Section 4.3.1.2). The feature desc is used to describe a structure whose root has been abstracted from some other structure. This is equivalent to a lambda operator, in permitting binding into an expression. For example, the phrase *the departure time*—which might be described
in lambda calculus as $\lambda x. debat_time(x)$—is represented in SIL thus:

$$
(5.3)
\begin{align*}
\begin{array}{c}
\text{id} : T \\
\text{type} : \text{time\_point} \\
\text{modus} : [\text{def} : \text{the}] \\
\text{id} : D \\
\text{desc} : [\text{type} : \text{depart} \\
\text{thetime} : [\text{id} : T]]
\end{array}
\end{align*}
$$

The two kinds of datastructure also differ in their implementation. Input and Output graphs are represented as Prolog terms, using the constructor ‘:’ to represent the feature:value relationship, and lists to represent bundles of features at a node. Re-entrancy is expressed by variable sharing, and when re-entrant terms are instantiated, by copying. The representation is that used by Gazdar and Mellish (1989), and is equivalent to what Bouma (1990) calls normal form. Networks in the discourse model are implemented using Nigel Gilbert’s object-oriented language NOOP. This uses the Prolog database for storing facts about nodes and roles. Inheritance is a built-in feature of the language. Initialisation takes the form of reading in the SIL definitions, and building the class inheritance hierarchy from the top down. Subsequently role constraints and inference rules are installed at nodes that have them. Correspondence between the discourse model representation and input/output graphs is established by using the discourse model node identifier as the filler of the id slot in the corresponding input or output graph.

SIL is also used to represent information at interfaces with the Task Module. Expressions of this kind differ from linguistically oriented input/output expressions in being flatter and involving considerably fewer primitives. This reflects more closely the way information is represented in databases. For example, the information about a journey from Paris is represented simply as in (5.4).

$$
(5.4)
\begin{align*}
\begin{array}{c}
\text{type} : \text{dbflight} \\
\text{thesourcecity} : \text{paris}
\end{array}
\end{align*}
$$

Such task-oriented forms do not appear in the semantic component of linguistic expressions. Instead, inference rules are used to establish equivalence between task-oriented and linguistic-oriented subgraphs in the discourse model. When required by some inference rule that structural representations of the two kinds be made equivalent, structure can be added to either kind of subgraph. For example, in Figure 5.6 constraints on the class single\_journey relate the departure and
arrival location information to attributes within the task object `DBFLIGHT1` (see also Figures 4.7, 4.8 and 4.9).

### 5.2.2 Interpretation

Interpretation incorporates two functions: stripping sub-structures from the representation which are not relevant to the discourse model; and anchoring the relevant sub-structures in the discourse model.

The semantic representation of the input is presented to the belief module in order to discriminate what it is capable of interpreting, and what is meaningless to it. The latter includes discourse markers, politeness expressions, attitude expressions and meta-referential expressions.

Material which can be interpreted needs to be anchored. This means finding a unique node in the discourse model to correspond to every node in the input graph. When the dialogue module passes the expression for interpretation to the belief module, it uses its knowledge of current goals and conversational structure to propose possible anchoring points. The success or failure of anchoring may be used by the dialogue module in refining its account of the conversational significance of the utterance. For example, the input may simply repeat earlier discourse material, or it may contradict what has been said previously; in both cases, the result is communicated via a status flag to the Dialogue Module (cf. Section 3.3.4).
The input is used to build new structure in the belief model, which mirrors closely its structure and content. Alternatively, it may fully or partly reference existing structure. If fully, no structure building takes place. Although corresponding closely to a subgraph of the discourse model, input/output graphs may be, as we have seen (Section 4.3.1.2) augmented with features indicating definiteness, pronominalisation, etc. When interpreting input representations, such features are used as signals to guide the search for an anchor point to an existing discourse entity. Once found, however, these features are not retained as part of the discourse model’s information.

An anchor point determines a node in the belief representation, in terms of a root index, and a path from that root. The anchor point is defined to be that unique object reached by following the path from the root. It may or may not exist previously as a discourse model object. Interpretation makes use of the procedure \textsc{interpret}_\textsc{dm}, which takes as input an input graph and an anchor point, and returns a set of bindings which anchor all nodes in the input graph. It takes place in three phases:

1. Establish a start point at the anchor point, creating a new object if necessary;
2. Start the (recursive) \textsc{interpret}_\textsc{dm} procedure from that anchor point;
3. Read off what (if any) new information has been written to any task objects, such as \textsc{dbflight1} in Figure 5.6.

When expressions include sufficient context, like \textit{I want to go to Paris}, locating an anchor is relatively easy. Shorter expressions such as \textit{five o’clock} require contextual information for their interpretation. An anchor can be found in one of following ways:

1. a context specified by the dialogue module can be used to guide search for anchors. In this case, the object or objects referenced by that context will be the candidate anchor. If several contexts are specified, it is possible to search each in turn until a call to \textsc{interpret}_\textsc{dm} has succeeded. However, it may be the case that no successful anchoring is possible given the proposed set of contexts.
2. For a given application, a set of classes exist, corresponding to events or relations the user is involved with. For the task **DBFLIGHT**, these include **GO**, **ARRIVE**, **DEPART**, **SINGLE-JOURNEY**. They are easy anchor points to find, because (relative to the current world and current task) every class has exactly one instance. These are treated as scenario-dependent inferrables (cf. Section 4.3.3.4), and as such assumed to be readily available to the hearer.

Once an anchor point has been established, **INTERPRET**\(_{DM}\) takes an index and a **SIL** expression, checks that the type of the expression corresponds to the class of the index, and then searches through the attribute-value pairs in the expression, checking that the attributes are permissible for the start object, and if so, recursively calling **INTERPRET**\(_{DM}\) for the values. Bottoming-out occurs when leaf nodes—slots containing simple string or integer values rather than typed objects—are reached.

Expressions containing the **desc** feature (corresponding, for example, to wh-questions or functional referential expressions such as *the time of departure* represented in (5.3)) are interpreted by interpreting first the subgraph at the **desc** role, then subsequently ensuring that the extracted object at the root of the whole expression has been found. Thus in Example 5.3, the node of type **depart** is first interpreted; this should lead not only to a binding for **D**, but one for **T** as well.

During anchoring new structure may be added to the current belief state. Any inferences resulting from this are then performed. Inference takes place by forward chaining; this has the advantage that any inconsistent states that may result from the application of inference rules to new information are discovered rapidly. By contrast, backward chaining might lead to inconsistencies remaining undiscovered.

World indices (cf. Section 4.3.4), or **views** are primitives of **noop**. All information about an object is recorded at some view **V**, and accessible only from views which inherit from **V**. Since a view represents an **extension** of a belief state, it is only necessary to store the information particular to that view; the rest can be retrieved by inheritance.

If part of the input contradicts existing data, a new **modification** view is created and the inconsistent information is written at that index. The dialogue module is
subsequently informed about the inconsistency, and the two conflicting views. A similar mechanism is used when two closely competing alternatives arrive in the input from the user—for example, in the case of uncertainty about whether Paris or Perros was spoken. For each alternative, a new view is created, inheriting from the current view. The dialogue module is then informed about the existence of this 'branching' in the discourse model, and all the (new) alternative world indices are returned.

Views are typed according to the way in which they relate to their parents. Typing of views affects their opacity properties, and how they are indexed. For example, if a modification taking place leads to the creation of a modification view $V_m$, $V_m$ inherits from the previous view $V_{old}$, and can be reached from that view via a link named modify. Other types of view are used for negation, alternatives and defaults.

Once the input expression has been fully interpreted, it remains for any task-oriented information resulting to be read off, and returned as the result of interpretation. This can be done because being forward-chaining, any inference rules which may have been used to derive task-oriented information (cf. the discussion on page 204) will have been applied.

5.2.3 The accessibility history and its use in interpretation

The accessibility history is represented as a global stack. Both input and output graphs are pushed onto the history. For input graphs, this takes place once anchoring is complete; for output graphs, once the current description graph has been fully determined. The history is potentially unlimited in size. However, the routines which access it are restricted to looking at the first $access_{bufsize}$ entries, where $access_{bufsize}$ is some appropriate predefined integer; more distant entries are then no longer available. Currently a value of 7 is used. This may seem implausibly large, but has the advantage that given entities remain accessible for longer. In order for the Example 4.47 (see page 153) to work, a length of at least three is required.

A shorter accessibility history is however advantageous, in terms of both memory
requirement and search effort. A compressed representation is therefore used: when a new entry $E$ is added, if a previous entry corresponds to all or part of $E$, then the previous entry is replaced by a back-pointer to $E$, together with a path locating the subtree if not all of $E$ is thus echoed. When searching the accessibility history, entries consisting of back-pointers need not be considered; as a result, accessibility routines are able to look beyond the first $access_{-}bufsize$ entries.

In addition, the set of the discourse entities currently represented in the accessibility history is maintained, ordered by recency. This can be used by routines which need to assess the relative accessibility of discourse entities, without searching through the accessibility history.

Entries in the accessibility history are labelled according to the view (or world index) at which they were introduced. This enables the use of the accessibility history to be restricted when there is an opaque relation between the current world and previous ones.

### 5.3 Planning utterances

I describe here the components of the implementation concerned with utterance planning, in particular the derivation of the description graph which is used to feed linguistic generation. Derivation of this graph is possible with or without the assistance of the accessibility history. Without it a certain amount of search is needed to ensure that the nodes of the description graph meets lexical admissability constraints. If substructures of previous description graphs are reused, search is reduced; moreover, an accessibility ordering on semantic entities corresponding to $\prec_h$ (cf. Section 4.3.3.2) can be derived as a result.

Before turning to consider in detail the derivation of the description graph, I give an overview of how it is related to the other stages of utterance planning. The decision to generate a system turn comprising one or more dialogue acts is taken by the Dialogue Module on the basis that this turn will serve to augment the dialogue history, in accordance with current communicative goals (cf. Section 5.1.2). As a result, it sends an utterance instruction to the Message Planner. This consists of a
Figure 5.7: Message planning: deriving the formulation instructions

list of dialogue act specifications, structured as in (5.5):

\[(5.5) \quad (\text{MoveIdentifier}, \text{DialogueAct}, \text{Content})\]

\textit{DialogueAct} is a dialogue act specification, consisting of an atomic dialogue act label specified by the relevant rule, together with optional features; the only feature of interest here being \([\text{repetition} : n]\), where \(n\) indicates the number of previous times the current act has been produced.

Figure 5.7 shows the stages of message planning. The message planner performs some optimisation, by merging together dialogue acts which relate to the same function, such as confirmation requests. The result is a list of templates. Each template is partly filled in by rhetorical rules. These associate fixed strings, root syntactic features, semantic matrix structures indicating propositional attitudes, or some combination of these, with the dialogue act labels. At the same time, for each template a \textit{description instruction} dependent on the rhetorical rule chosen, and containing the \textit{Content} component of the dialogue act specification, is sent to the Belief Module in order for the description graph to be derived. This is then merged.
with the template. The set of fully specified templates, or formulation instructions, is forwarded to the Linguistic Generator.

5.3.1 Deriving the description graph

A description graph is a subgraph of the current state of the discourse model, according to some specified view. It is a directed acyclic graph (DAG) with a single root, and has two fundamental properties:

1. Because it is a subgraph of the discourse model, it is true of the model (in the chosen view);

2. It is lexically realisable: that is, it is capable of being derived compositionally from linguistic structures specified in the lexicon.

A further desirable property is that the interlocutor should have no difficulty in attaching the structure within his own copy of the discourse model.

Description graph derivation may be complex. Not all substructures in the discourse model are lexically realisable. Structures also occur that serve to instantiate necessary inferential links, or that relate directly to datastructures used in the Task Module. This means that in general, the set of linguistically realisable subgraphs is a proper subset of the set of all subgraphs. Because of the modular nature of the system, it is not possible (nor necessarily desirable) to access the lexicon in determining which subgraphs can be description graphs. Instead, when the lexicon is compiled, every local node configuration that forms part of a semantic field of some entry, is stored in a lexical constraints file. Consider for example the lexical entry for travel, which takes as arguments a source, a destination and a date. This is shown in (5.6).

```
(5.6)  
lexeme : travel  
  head : major : v  
  syntax :  
    args :  
      id : T  
      type : go  
      thesource : Src  
      thegoal : Dest  
      thedate : Dat  
    semantics :  
      [head : major : prep(from)]  
      [head : major : prep(to)]  
      [head : major : prep(on)]  
  semantics : Src  
  semantics : Dest  
  semantics : Dat
```
In the actual lexical entry, the variables \textit{Src}, \textit{Dest} and \textit{Dat} are further specified for their relevant semantic types. On the basis of the semantics field of this lexical entry, it is possible to derive a local node constraint on description graphs, stating that a node of type \textit{go} is required to have the roles \textit{thesource}, \textit{thegoal} and \textit{thedate}, and only those. The constraint is defined at the class level, on the type that is root of the local configuration. If there exist more than one lexical entry that defines a such constraint on a class of entities, as would be the case for the verbs \textit{go}, \textit{travel}, \textit{fly} which use variants of \textit{go} as their semantic root, then the constraint on description graphs is relaxed to allow a choice of possible lexical constraints to apply at such a node. A lexical constraint is described in terms of a node, together with a list representing the set of paths which must be followed in the graph, for the constraint to apply. Paths are represented as lists of links, so the constraint for \textit{go} derived from the lexical entry (5.6) can be represented:

(5.7) \quad \text{lexical\_constraint(go, }[[\text{thesource}], [\text{thegoal}], [\text{thedate}]]).

These lexical constraints are read in at initialisation time, when they become attached to class nodes.

The remaining constraint on a description graph is that it should be uniquely attachable by the interlocutor, without excessive search. This will be the case if the root has been mentioned or is inferrable, and the contents describe it uniquely.

I now describe the basic procedure for deriving a description graph. The input may be a combination of views and task-oriented descriptions; I shall deal with the simple case, where only a task-oriented description is provided. This is specified as:

(\textit{TaskId}, \textit{AVPairs}),

where \textit{TaskId} identifies a task object, and \textit{AVPairs} is a set of pairs each consisting of an attribute label, followed by a value which is string- or integer-valued, and doesn’t itself correspond to a node in the discourse model. The procedure \texttt{find\_ids} is a function that takes this input, and returns a set of core nodes \textit{CIds}, each of which
is potentially lexically realisable, by conforming with some lexical constraint. This is done by asserting the task-oriented description in the discourse model (where it may already be present) and collecting those semantic objects which are necessarily present when the task information is present, according to the inference rules.

_Clds_ may already constitute a *convex set*—that is, a set of objects which uniquely determine a subgraph of a description. Failing this, they may at least be bundled into a set of convex sets, and the subgraphs for these found. The procedure _describe_ids_ takes a set of object identifiers, and returns a set of such subgraphs. It does this by selecting a potential root _Id_r from the set, one that dominates at least some of the remaining elements. It runs the _describe_from_root_ routine (described below) starting with _Id_r and ensuring that the only nodes included in the graph are taken from _Clds_. Next, _describe_ids_ iterates over those elements of _Clds_ which have not been included in some subgraph, until all of _Clds_ are consumed. The resulting set of core graphs, _CGs_, covers all the elements of _Clds_; ie, the union of the nodes in _CGs_ is a superset of _Clds_.

The procedure _describe_from_root_ may be described as a function which takes a semantic identifier, _Id_r, and traverses the discourse model to derive a graph with root _Id_r_. This derivation is recursive: daughter nodes for _Id_r are found, and _describe_from_root_ is applied at each of these nodes, until atoms or objects with no information attached are found. Only lexically admissible roles are followed. A copy of the lexical constraint chosen is used to keep a tally of those roles found; the constraint is thus checked incrementally. whether or not the constraint has applied can be verified.

In order to determine the required extent of a description graph, the notion of *attachability by the interlocutor* is used. It is assumed that the interlocutor applies principles similar to the system's, in determining an attachment point. Mutually known shared contexts and task objects can be used as they were for interpretation; the attachment process must take a version of the description graph with anchoring information removed, and correctly predict the anchoring. For example, the graph deriving the utterance *two thirty* in Example 4.47:A2, might be the following:
This minimal graph is attachable by the interlocutor if an unanchored copy—i.e., one in which the values of the \textit{id} feature are replaced by variables—can be anchored to give correct bindings for these variables, given the current shared context.

If \textit{CGs} is a singleton set whose root is attachable for the interlocutor, this becomes the description graph returned. If not, it is necessary to chain upwards to some attachable node which both spans all the subgraphs. Upward chaining as implemented is not particularly efficient. The procedure \texttt{FIND\_SPANNING\_ROOT} starts with the roots of the members of \textit{CGs}, \textit{CRs}, and returns a node \textit{Id}, which is the root of a graph that includes the nodes in \textit{CRs}. The procedure works by selecting nondeterministically \textit{Id} \textsubscript{i} \in \textit{CRs}, and calling the procedure \texttt{CHAIN\_UP} to derive a chain of role links from \textit{Id} \textsubscript{i} back up to some \textit{Id}, by following lexical back-links (explained below). An attempt is then made with \texttt{CHAIN\_UP} to reach this \textit{Id} from each of the other members of \textit{CRs}. Failing this, backtracking will lead to a node \textit{Id} \textsubscript{i}, which is higher up, and search is resumed.

Lexical back-links are derived from the lexical constraints at initialisation time. They specify, for a given class, the role link link in the discourse model which may be followed backwards from an instance of that class to some dominating node. For example, because of the lexical constraint for \textit{go} described in (5.7), the following back-links can be added to the class \texttt{LOCATION}:

\begin{align}
(5.9) \quad \texttt{LOCATION} & \leftarrow (\textit{thesource}) \texttt{go} \\
(5.10) \quad \texttt{LOCATION} & \leftarrow (\textit{thegoal}) \texttt{go}
\end{align}

Thus starting with an object \textit{id}_{loc} of type \texttt{LOCATION}, \texttt{CHAIN\_UP} can look in the discourse model to find if there is any object of type \texttt{go} from which \textit{id}_{loc} may be reached, via a role link labelled \textit{thesource} or \textit{thegoal}. Of course, there may be other lexical back-links to try.

214
The success criterion for \texttt{chain\textunderscore up} is currently very simple. The node at the top of the chain must be known to the interlocutor, and unique of its class. A more sophisticated success criterion is discussed in the next section.

The spanning root together with \textit{CGs} may not be sufficient to determine a description graph that meets the lexical constraints. The situation is illustrated in Figure 5.8. Given the root \texttt{Id$_*$}, and core subgraphs $C_1 \ldots C_n$ headed by \texttt{Id$_{c1} \ldots Id$_{cn}$}, there may still exist intermediate nodes \texttt{Id$_1, Id$_2$}, \ldots whose lexical constraints haven't been fully satisfied. Additionally, \texttt{Id$_*$} itself may not fully meet any of the lexical constraints for its class. Nodes which do not fully satisfy lexical constraints are represented in the figure by open circles. Starting with this \textit{undersaturated graph} then, the root node and each intermediate node are tested for satisfaction of some lexical constraint. For each node \texttt{Id$_j$} where satisfaction is not possible, a constraint satisfied by the existing branches is chosen, and a variant of \texttt{DESCRIBE\textunderscore FROM\textunderscore ROOT} is used that starts at \texttt{Id$_j$}, and adds subgraphs to its existing subgraphs, until the constraint is met. The resulting saturated spanning graph is returned as the description graph.

As an example, consider the request to derive a description graph for the input in (5.11). The procedure \texttt{FIND\textunderscore IDS} takes this input, and returns \texttt{CIds}: see (5.12). \texttt{DESCRIBE\textunderscore IDS} then selects \texttt{location2} from \texttt{CIds}, and calls \texttt{DESCRIBE\textunderscore FROM\textunderscore ROOT} with this, to find that the first member of \textit{CGs} (5.13) can be derived, thereby consuming \texttt{location2} and \texttt{city2}. The remainder of \texttt{CIds} is treated in the same way,
Figure 5.9: Searching for a spanning root: unsuccessful and successful cases to derive the second member of CGs.

\[(5.11)\quad \langle \text{dbflight1}, \{\text{depcity} : \text{london}, \text{deptime} : 1715\}\rangle\]

\[(5.12)\quad \text{CIds} = [\text{LOCATION2, CITY2, TIMEPOINT1, HOUR1, MINUTES1}]\]

\[(5.13)\quad \text{CGs} = \langle \begin{bmatrix} \text{id} : \text{location2} \\ \text{type} : \text{location} \\ \text{thecity} : \begin{bmatrix} \text{id} : \text{city2} \\ \text{type} : \text{city} \\ \text{value} : \text{london} \end{bmatrix} \end{bmatrix}, \begin{bmatrix} \text{id} : \text{timepoint1} \\ \text{type} : \text{timepoint} \\ \text{thehour} : \begin{bmatrix} \text{id} : \text{hour1} \\ \text{type} : \text{hour} \\ \text{value} : 17 \end{bmatrix} \\ \text{theminutes} : \begin{bmatrix} \text{id} : \text{mins1} \\ \text{type} : \text{mins} \\ \text{value} : 15 \end{bmatrix} \end{bmatrix} \rangle\]

\[(5.14)\quad \text{CRs} = [\text{location2, timepoint1}]\]

Since there is more than one member of CGs, \text{FIND\_SPANNING\_ROOT} is called, starting with CRs: see (5.14). Figure 5.9 illustrates some of the search that ensues. The back-link of (5.9) succeeds in linking location2 to go1; since the latter node meets the accessibility constraint, it is returned as the result of chain_up. However, attempts to use chain_up to link timepoint1 to go1 fail, since there is no lexical constraint for go with a time-slot. So go1 is eliminated as a spanning root, and
another candidate tried, until \texttt{chain\_up} returns the candidate root \texttt{departure1},
which is linked to \texttt{location2} in the discourse model via the single chain \texttt{[theloc]}. On this occasion the root can also be reached from \texttt{timepoint1}, via the link \texttt{[thetime]},
and \texttt{find\_spanning\_root} succeeds. The graph derived however is undersaturated,
because the relevant lexical constraint for \texttt{depart} is that of (5.15), where thetheme can
be specialised to contain an object of type \texttt{individual} or \texttt{carrier}.\footnote{Strictly speaking, thetheme is a role on input/output graphs, needed for economy of lexical
description. It acts as an overloaded function, since in the discourse model it is able to map onto
either individuals or carriers. More details of this are given in Section 5.7.1.}
In this case calling \texttt{describe\_from\_root} for the role thetheme produces a subgraph that
refers to the user. The resulting saturated description graph is shown in (5.16).

\renewcommand{\arraystretch}{1.0}
\begin{equation}
\text{lexical\_constraint(\texttt{depart}, [[\texttt{theloc}], \texttt{[thetime]}, \texttt{[thetheme]]})}
\end{equation}

\begin{equation}
\begin{array}{c}
\text{id} : \texttt{departure1} \\
\text{type} : \texttt{depart} \\
\text{theloc} : \begin{array}{c}
\text{id} : \texttt{location2} \\
\text{type} : \texttt{location} \\
\text{thecity} : \begin{array}{c}
\text{id} : \texttt{city2} \\
\text{type} : \texttt{city} \\
\text{value} : \texttt{London}
\end{array}
\end{array} \\
\text{thetime} : \begin{array}{c}
\text{id} : \texttt{timepoint1} \\
\text{type} : \texttt{timepoint} \\
\text{thehour} : \begin{array}{c}
\text{id} : \texttt{hour1} \\
\text{type} : \texttt{hour} \\
\text{value} : \texttt{17}
\end{array} \\
\text{themins} : \begin{array}{c}
\text{id} : \texttt{mins1} \\
\text{type} : \texttt{mins} \\
\text{value} : \texttt{15}
\end{array}
\end{array} \\
\text{thetheme} : \texttt{USER}
\end{array}
\end{equation}

As a result, an utterance such as you want to leave London at 17:15 may be
produced.

\subsection*{5.3.2 Description with the accessibility history}

The algorithm for getting a description graph can be modified, by replacing the
procedures \texttt{describe\_ids} and \texttt{find\_spanning\_root} with \texttt{describe\_ids\_ah} and
\texttt{find\_spanning\_root\_ah}. These procedures represent the same functions as their
predecessors: that is, \texttt{describe\_ids\_ah} maps CIds to CGs, and \texttt{find\_spanning\_root\_ah}
finds a spanning root for CRs.
DESCRIBE\_IDS\textsubscript{ah} takes an additional argument which is initially the current accessibility history. If this is empty, then the default is to apply DESCRIBE\_IDS. Otherwise, the first graph in the accessibility history is taken to be CurrentGraph; if there is some subset \textit{Id\_s} of \textit{Cl\_ds} which is exactly covered by some subgraphs \textit{SG\_s} of CurrentGraph then the procedure is called recursively with the remainder of \textit{Cl\_ds} and the tail of the accessibility history; \textit{SG\_s} is added to the result. Otherwise, if no subgraphs of CurrentGraph are suitable, then CurrentGraph is ignored, and \textit{Cl\_ds} remains intact for the recursion. If some \textit{Cl\_ds} remain when all of the accessibility history has been tried, the default DESCRIBE\_IDS is used.

The motivation for using DESCRIBE\_IDS\textsubscript{ah} instead of DESCRIBE\_IDS is that the former only needs to look up already built subgraphs from a limited set, in order to fuse together \textit{Cl\_ds}. If the accessibility history is not used, however, it is necessary to perform search in the discourse model, constantly checking the lexical constraints. Similarly the procedure FIND\_SPANNING\_ROOT, which finds a spanning root for a set of subgraphs, has considerably less work to do if that root can be found in the accessibility history.

For each member of \textit{CG\_s} found in the accessibility history, DESCRIBE\_IDS\textsubscript{ah} records its entry number. FIND\_SPANNING\_ROOT\textsubscript{ah} works like this: if every member of \textit{CG\_s} belongs to the same accessibility history entry \textit{AH\_i}, then the lowest node in \textit{AH\_i} that dominates them all is the spanning root, \textit{Id\_s}. Moreover, the graph of which \textit{Id\_s} is the root is saturated, and is therefore the description graph required. If a majority (but not all) of \textit{CG\_s} belong to \textit{AH\_i}, \textit{Id\_s} can be found for these. Then the procedure which searches in the discourse model, FIND\_SPANNING\_ROOT can be applied to verify if \textit{Id\_s} is a spanning root for the remainder of \textit{CG\_s}. This situation is illustrated in Figure 5.10. Here the accessibility history is shown as an overlay of the discourse model. The core graphs \textit{CG\_1} and \textit{CG\_2} have been found in the entry \textit{AH\_1}, together with a spanning node \textit{Id\_s}. A search is then made in the discourse model to find if the root of third core graph, \textit{CG\_3} is capable of chaining up to \textit{Id\_s}.

Alternatively, the structure of a previous graph may still be used, if the graph built by FIND\_SPANNING\textsubscript{ah} turns out to be structurally parallel to some antecedent entry, as defined in (5.17):
Two graphs $G_1$ and $G_2$ are *structurally parallel* if their roots are of a compatible type, and there is a non-empty set of paths $Paths$ such that

$$\forall Path \in Paths \bullet G_1 : Path \text{ is structurally parallel to } G_2 : Path$$

where given a graph $G$ and path $Path$, $G : Path$ is defined to be $G : root : Path$, ie the subgraph of $G$ obtained by following $Path$ from the root of $G$.

A previous entry $AccHist[n]$ is potentially structurally parallel to the graph being built, if a number of its subgraphs match members of $CGs$, $CGs_0$. The procedure `tandem_possible` tests for potential structural parallelism, by taking a set of core graphs which need to be incorporated in the target graph, and a previous entry. If successful, it returns the set of paths $Paths$ which will locate the core graphs with respect to the root of the new graph. The procedure `find_spanning` is then run on $CGs_0$, using the paths in $Paths$ to guide it. The resulting graph is true in the discourse model at the current (description) view, and is structurally parallel to $Acchist[n]$. Any remaining subgraphs (in $CGs - CGs_0$) are then attached.

If neither a reusable entry, or a structurally parallel entry can be found the default procedure `find_spanning` is applied.

Normally, both `find_spanning` and `find_spanning withheld` return a single graph. However either procedure is capable of returning a *tandem pair*, consisting of two or more graphs that are structurally parallel. If possible these should contain a common subgraph; this is necessary for conjoined descriptions that are to be realised as VP-conjunctions: $BA123$ leaves London at five thirty and arrives in Paris at six
Derivation using the accessibility history as described here can reduce search and verification in the discourse model. This works well unless violation of monotonicity has taken place during the period covered by the current accessibility history. If this is the case, graphs and subgraphs taken from the accessibility history may not correspond to the information in $DM_{CV}$, the state of the discourse model in the view $CV$ for which the description is required. For this reason, graphs in the accessibility history are prefixed with the identifiers of the views that were current at the time they were entered (Section 5.2.3). If $\text{find-spanning}_{ah}$ uses an entry (or subentry) prefixed by $CV$, there is no problem. Similarly, if the prefixing view $PV$ is an ancestor of $CV$, and this ancestry is transparent, what is true in $PV$ will also be the case in $CV$, so the entry can be used. If on the other hand $PV$ and $CV$ do not have a transparent accessibility relationship, any solution found must be verified in the discourse model. This still involves less work than the search to find the solution would.

5.3.3 Contributions to focal assignment at the discourse level

Two aspects of focal information may be obtained at the description level. Intentional focus is the result of some discourse elements becoming marked by virtue of their appearance in the original description instructions. Discourse focus, or attentional focus, provides an ordering over some discourse elements, based on their actual or assumed accessibility. Information about these aspects of the description graph is somewhat orthogonal to that contained in the graph, since it is indifferent to graph structure. The description graph is therefore augmented with an information profile $IProfile$. This is composed of three elements, shown in (5.18). The output of the description routine is therefore the datastructure $DescOut$ described in (5.19).

\[ IProfile = (\text{IntentionalFocus}, \text{DiscourseOrdering}, \text{ReuseInstructions}) \]
In this section I discuss the production of *IntentionalFocus*, *DiscourseOrdering* and *ReuseInstructions*.

Intentional focus (cf. Section 4.4) is that marked on some elements of an utterance because inclusion of those elements was part of the decision to produce that utterance. In fact the description instructions do not include nodes in the discourse model, but task-oriented parameter-value pairs. However, as a result of \( \text{FIND\_IDS} \), trees corresponding to the information in the initial instructions are constructed or retrieved from the accessibility history.

The feature *ifocus* is thus defined on some of the nodes in the description graph. In Section 4.4 I suggested that, apart from usually ensuring prominence of the nodes to which it applied, the kind of prominence, together with any local accent configuration to be realised on the surface form, was largely dependent on the dialogue act in question. For example, an unauthorised confirmation will tend to emphasise the modified elements, with the possible addition of a deferential fall-rise contour. Intentional focus may thus be treated as a hook specifying only a position with respect to the rest of the utterance, and onto which dialogue-act-specific features may be added. This is appropriate when all intentionally marked elements are marked similarly. Where this is not the case, for example, for the questioned element of an \((\text{init}; \text{query})\), or to distinguish *Old* and *New* in a correction \((\text{correct}; (\text{Path}, \text{Old}, \text{New}))\), an optional additional feature may be used to single out the nature of the parameter with respect to the dialogue act.

A further type of intentional focus depends on the speaker presenting information as a functional relation, as discussed in Section 4.3.4.3. Although such a relation may be derived from some necessary property of the database (which can be assumed to be common knowledge) the speaker may choose to present arbitrary information using intonationally marked themes and rhemes. I therefore term such marking *rhetorical focus*. The implementation presented here does not enter into details of user-modelling or argumentation; I do not therefore discuss how such a marking is derived. The marking is represented conceptually as a dependency between semantic
This states that if a discourse object of type move (or a subtype) has an object place, at path [theplace], and an object s_time; at path [thetime, thetime], then there is a rhetorical dependency between place and s_time, which can be linguistically represented using theme-rheme marking. In the case of the utterance BA123 leaves Paris at twelve thirty this will result in Paris being marked as theme, and twelve thirty as rheme. In the absence of an argumentation component, rhetorical dependencies are produced under the control of the implementor using the command put_rh_depend. An example of the working of this is included in Appendix D, pages 341–343, where the rhetorical relation between place and time is applied to the pair

\[(Paris \ Charles de Gaulle, eight twenty seven).\]

It was proposed in Section 4.3.3 that attentional focus be represented as a prominence ordering over discourse elements. Using the accessibility history in the derivation of the description graph allows as a side effect the prominence ordering \(\prec_h\) to be computed. These are defined in the following way. I assume that the set of integers EntryNos is used to index corresponding pairs of \((AH_i, LH_i)\) taken from the accessibility and linguistic histories. Let the nodes of the description graph \(DG\) be \(E_{DG}\). Instructions is the set

\[
\{ (i, \text{inst}) \mid i \in \text{EntryNos} \land \text{inst} \in \{ \text{eq, neq, mod, par} \} \} \cup \{ \text{new} \}
\]

The function \(Instruction_{DG} : E_{DG} \rightarrow \text{Instructions}\) is defined as follows:

1. if \(e_r \in E_{DG}\) is the root of a subgraph identical to one in some entry \(AH_i\) \((i \in \text{EntryNos})\), then \(Instruction_{DG}(e_r) = (i, \text{eq})\);

2. if \(e_r \in E_{DG}\) is the root of a subgraph \(SG_r\) corresponding to a subgraph \(SG_r'\) with the same root but not identical, in \(AH_i\), then \(Instruction_{DG}(e_r) = \)

\(^2\)Note that the repeated label in the path [thetime, thetime] is permissible, since the first role is in respect to move, and the second in respect to time_point. See Figures 5.4 and 5.5.
\( \langle i, \text{mod} \rangle \);

3. if instead \( SG_r \) and the corresponding subgraph \( SG'_r \) are structurally parallel but have different roots, then \( \text{Instruction}_{DG}(e_r) = \langle i, \text{par} \rangle \);

4. if \( SG_r \) and \( SG'_r \) are as in case 2, and \( e_t \) and \( e'_t \) are nodes in the same relation to the root in both graphs, but are different, then \( \text{Instruction}_{DG}(e_t) = \langle i, \text{neq} \rangle \);

5. if \( SG_r \) and \( SG'_r \) are as in case 2, and \( e_t \) is the root of a subgraph of \( SG_r \) which does not appear in \( SG'_r \) then \( \text{Instruction}_{DG}(e_t) = \langle \text{new} \rangle \).

Reuse instructions are assigned as follows:

- The roots of all subgraphs belonging to a previous entry \( \text{AccHist}[n] \) obtain instruction \( \langle n, \text{eq} \rangle \);

- In the case of previous \( \text{AccHist}[n] \) parallel to the current description \( DG \), the root of \( DG \) gets the instruction \( \langle n, \text{mod} \rangle \) or \( \langle n, \text{par} \rangle \), depending on whether or not it is the same as the root of the graph found in \( \text{AccHist}[n] \);

- Subgraphs of a graph \( DG \) which is structurally parallel obtain reuse instructions \( \langle n, \text{eq} \rangle \), \( \langle n, \text{mod} \rangle \) or \( \langle n, \text{neq} \rangle \), depending on whether the corresponding subgraph is identical, structurally parallel with the same root, or different;

- Additions made to a graph are similarly marked; the instruction \( \langle \text{new} \rangle \) being used where the new graph adds to rather than changes the structure already built;

- Remaining nodes are assigned by default reuse instructions reflecting their accessibility.

Marking nodes rather than graphs is important, because the markings on the nodes of a graph will not in general be the same as that on its root.

The relation \( \prec_h \) on nodes of a description graph \( DG \) may be defined in terms of the reuse instructions derived:

\[
(5.21) \quad \{ \text{Mod, Eq} \} \prec_h \text{REST} \prec_h \{ \text{Par, Neq} \}
\]

223
where
\[
\begin{align*}
Mod &= \{ N \in \text{Nodes}(DG) \mid \text{reuse}(N, (\_\_\_, \text{mod})) \} \\
Eq &= \{ N \in \text{Nodes}(DG) \mid \text{reuse}(N, (\_\_, \text{eq})) \} \\
Par &= \{ N \in \text{Nodes}(DG) \mid \text{reuse}(N, (\_\_, \text{par})) \} \\
Neq &= \{ N \in \text{Nodes}(DG) \mid \text{reuse}(N, (\_\_, \text{neq})) \}
\end{align*}
\]

(5.22)

Here and elsewhere I follow the Prolog practice of using "\_" to represent an anonymous existentially-quantified variable.

Other components of the relation \(\prec_{DM}\) are arrived at by considering inferrability, deixis, and informativeness, as follows:

**Inferrability:** Any node introduced into the discourse model as a result of inference, during either interpretation or description, is marked with the feature [owner: inferred]. Subsequently, if this node occurs in an input or output graph, the value of owner is revised to whoever spoke: system or agent, or to shared if the node has been included in graphs produced by both parties. Any node with [owner: inferred] which occurs in \(\text{Nodes}(DG)\) is therefore defined to be a member of the set \(\text{Inferrables}_{DG}\).

**Deixis:** Currently the Belief Module only treats two distinguished elements as deictic, those corresponding to system and user.

**Informativeness:** Informativeness due to some nodes being inherently informational is handled simply using the class hierarchy. In the current definition of SIL, types associated with names, times, dates, flight numbers and places are subsumed under the generic node property, and apart from logical operators and discourse markers exhaust this category. Nodes subsumed by property, with those exceptions, are defined to be relatively informative: [informative: +].

Determining the relative informativeness of, say, airport names over city names, cannot be done on the basis of the information in SIL alone, since it depends on the nature of facts in the database. What is required is meta-information of the kind: 'the relation between city and airport is one-to-many'. Statement of such
a relation must be circumscribed by the kind of structure to which it should apply. Since only one-to-many relations are of interest in computing relative informativeness, I therefore specify instances of the relation \texttt{ONE\_TO\_MANY/3} as exemplified in (5.23–5.25).

\begin{align*}
(5.23) \quad & \texttt{ONE\_TO\_MANY(location, [thecountry], [thecity])} \\
(5.24) \quad & \texttt{ONE\_TO\_MANY(location, [thecity], [theairport])} \\
(5.25) \quad & \texttt{ONE\_TO\_MANY(location, [thecity], [thestation])}
\end{align*}

For a given location (which has possible attributes \texttt{thecountry, thecity, theairport, thestation}) many cities will in general belong to a single country, and so on. The rules given are defaults, and exceptions are easy to find, especially in the case of cities that have only one or no airports or stations. More specific relation instances need to be defined in those cases. A binary relation \texttt{More\_Informative} can then be defined on selected pairs out of \texttt{Nodes(DG)} according to whether they instantiate some definition of the \texttt{ONE\_TO\_MANY} relation. In the example dialogues of Appendix D this rule is applied during ordering, to give airports relatively higher prominence than city names (see for example page 331).

The relative underinformativeness of elements denoting propositional attitudes is not handled by the description routines. Since these elements do not appear in the Information Profile, they are by default marked low in the overall prominence order.

In this section I have described how focus properties of discourse elements belonging to the description graph are produced during the description planning stage of generation. How these are combined into a prominence ordering and realised as annotations on text sent to the rule-based-synthesizer is the subject of Sections 5.5.2–5.5.3.
5.4 Linguistic generation

On the basis of the formulation instructions (cf. Figure 5.7) the Linguistic Generation module produces a linguistic representation suitable for speech synthesis. The components responsible for producing a full analysis tree are shown in Figure 5.11. Section 5.4.1 describes the formalism used for representing linguistic knowledge, and how the two linguistic knowledge bases are compiled. In Section 5.4.2 I explain the head-driven generation algorithm. Section 5.4.4 gives details of the re-use algorithm. Prosody generation is described in the following section, 5.5.

5.4.1 Representing linguistic knowledge

The UCG formalism was presented in Section 4.2.2. An example lexical entry is given in (5.26):
In addition to the basic phonology, syntanz, semantics, the sign has fields for slash and order. The former (not shown in 5.26) is used in the gap-threaded treatment of movement (see Section 5.4.3 below). The latter is instantiated in argument categories, to indicate their surface position with respect to the functor (head).

The grammar used here is different from standard uccg in two respects. Firstly, domain-specific semantics is incorporated in the lexicon to a much greater extent. The semantic type can take any value from the subsumption hierarchy defined in sil. The type arrive in (5.26) is subsumption-compatible with event; therefore the entry can be inserted into an environment which requires a sign of this semantic type. The subcategorisation frame can thus be simultaneously syntactically and semantically constraining. Secondly, as explained in Section 4.2.2, the rules of functional application:

\[
A/B \rightarrow A \\
B A/B \rightarrow A
\]
have been elaborated to permit underspecified order, adjacency and optionality.

In (5.26), \textit{theplace} argument is optional and can occur in any position after the functor.

The lexicon is defined using the \textsc{datr} language (Evans and Gazdar 1989). Starting with a set of \textit{definitional sentences} in which redundancy and reduplication is kept to a minimum, the principles of inheritance with defaults are applied to derive a lexicon with rich structure-sharing both within and among entries. The lexicon is compiled out of the \textsc{datr} sentences offline (Andry et al. 1992).

The lexical entries for generation and parsing incorporate the same knowledge, being derived from the same source. However the generation lexicon differs from the parsing lexicon in its compiled form for two reasons. Firstly, the generation lexicon is indexed principally by the \textit{semantics} field. Entries are indexed by semantic type, and ordered according to the amount of semantic detail they are capable of absorbing from the input description. This is done by grouping entries according to their semantic type, and sorting groups on the basis of subsumption. The second difference is that a \textit{morphological lexicon} is compiled out separately from the lexicon of base forms. The entries in this are ordered according to specificity, lexical exceptions—ie, irregular forms—being first; each entry serves to map an uninflected form onto a surface form, given a certain specification of syntactic features. Additionally, a set of \textit{lexical constraints} for use in discourse model initialisation are produced (cf. Section 5.3.1).

\section*{5.4.2 Head driven generation}

The \textit{head-driven bottom-up generation} algorithm (vanNoord 1990) is particularly suitable for lexicon-based grammars such as the one used in Sundial. It also has the advantages of being goal-directed (hence of greater efficiency), and not limited by the recursion problem present in top-down algorithms.

The algorithm, in simplified form, can be described as the process of finding a lexical candidate for the input sign whose semantics unifies with that of the input, then recursively generating its arguments. However, a so-called \textit{lexical pivot}, chosen
on the basis of its semantics, does not guarantee syntactic coherence. For example, given the srt description in (5.27):

(5.27)

\[
\begin{array}{l}
\text{id : gol} \\
\text{type : go} \\
\hline
\text{id : location2} \\
\text{type : location} \\
\text{thesource : } [ \\
\text{id : city2} \\
\text{type : city} \\
\text{value : london} ]
\end{array}
\]

the algorithm generates: 3

(5.28)

\[
\text{travellondon : v : go} \\
\text{travel : v : go} \\
\text{london : n : location }\rightarrow
\]

where \text{london} is syntactically an NP but is required by the subcategorisation frame of \text{travel} to be a prepositional phrase. To deal with this, I introduce \text{specifiers}, which simply act as category raisers, and \text{modifiers}, which may add material to the semantics while maintaining the correspondence between goal semantics and pivot semantics (Bouma 1988). Either of these will be referred to as \text{raisers}. In the analysis tree shown in (5.28), the raiser \text{prep(from)/n} is required above \text{london}.

The function \text{generate} in (5.29) takes as input a sign \text{sign?}, of which only the semantics need be specified, and produces an analysis tree \text{treeout!}. It requires that the arguments in the subcategorisation frame of the candidate lexical entry \text{lexcand} are recursively generated first; after this an attempt is made to find a chain of raisers which will raise the semantics and the syntax of the result of applying \text{lexcand} to \text{Args}, to meet the input constraints.

---

3The analysis trees shown here and elsewhere represent a condensed form of the feature structures actually generated. The semantics (shown in small capitals) in particular shows only the semantic type of the constituent. Order with respect to the head is shown by arrows; the lexical head of every node is shown immediately below the sign for the node, and does not have an order marking.
The algorithm is described here using a variant of the ‘Z’ language (Spivey 1988), a notation which uses set-theoretic constructs and predicate calculus in a structured way to build formal specifications. The definition for generate is structured in terms of a set of declarations, followed by sentences which must become true during the process of generation. In order to specify clearly that unify exports its bindings to the environment, the bindings $\theta$ are shown returned as the result of unify, and applied using the operator ‘o’.

The types used in the specification are as follows:

- $\text{SIGN}$: the set of all well-formed signs, as defined above
- $\text{LEX}$: the subset of $\text{SIGN}$ corresponding to lexical entries
- $\text{RAISER}$: the set of raising categories

Because the lexicon is subsumption-ordered, lexical access is relatively efficient. For a given type, the entries which are most specific—either because of their semantic specificity, or the specificity of their subcategorization requirements, or both—are tried first. Should these prove too specific, less specific entries are then tried.

The operation raise matches the sign raiser against the chain of lexical raisers which make it up, recursively raising lexcand as it goes. Morphological instantiation is delayed until this point, to avoid the possibility of backtracking that might arise.
from instantiating a feature structure prematurely. For example, in the case of from london, a recursive call to generate has found unraised to be the sign london : n : location. In this case the operation raise consists of simple functional application with respect to the raising sign from : prep/n.

With raising, instead of (5.28) the correct form given in (5.30) is generated:

(5.30)

\[
\begin{align*}
\text{travelfromlondon} & : v : \text{GO} \\
\text{travel} & : v : \text{GO} \\
\text{fromlondon} & : \text{prep} : \text{LOCATION} \rightarrow \\
\text{from} & : \text{prep} : \text{LOCATION} \\
\text{london} & : n : \text{LOCATION} \rightarrow
\end{align*}
\]

The recursive call to generate requires \( \text{sign?} : < \text{syn head} >= \text{prep(from)} \), as specified in the entry for travel. Since lexcand has category \( n \), the raiser is required.

5.4.3 Refinements

5.4.3.1 Moved constituents

The basic algorithm assumes that while the input \( \text{sign} \) may be partially specified for syntactic features, it has no specified arguments. This is not always the case. For example, in the case of an ‘equi’ verb such as want, the input sign for the recursive call to generate which will produce the embedded verb-phrase, already has its subject argument instantiated. It is therefore necessary to match such arguments against those of the candidate lexical entry lexcand, but ignore them in testing for instantiation and in recursively generating. For example, with do you want to travel to Paris from London, the subject of travel, which is specified in its args list, is not generated for this reason.

Moved constituents are handled by a threaded argument to generate constraining the nature of any expected gap (eg. Pereira and Shieber 1987). Gaps are introduced by \( wh \) constituents, or by non-lexical rules such as topicalisation (eg. Gazdar et al. 1985). A gap is threaded through the generation tree, on the basis that most lexical constituents, and hence most phrasal constituents, simply pass the
gap on. So-called 'island constraints' on movement may be enforced by specifying in the lexicon that certain phrasal categories cannot pass gaps. A gap is eliminated, leaving instead of an expected constituent a placeholder or trace, when the constraints of that constituent match the constraints of the gap. An example of such a structure is given in Section 5.7.2.

5.4.3.2 The surface string

The surface string for a sign is built by applying the rules of categorical combination to its head and arguments. In order for a constituent's string to be generated, and hence be capable of being spoken, all strings corresponding to subconstituents must be complete. This places a constraint on the possibility of using the algorithm within an incremental production architecture. Where order of subconstituents is unspecified, the string is produced using the default order of the lexical entry.

During traversal and building of the analysis tree, the components of the surface string are stored as the feature phon at every level. When the lexical head has been selected, a skeleton structure for the phrasal string is put in place, with placeholders for components not yet generated. As generation of the arguments unfolds, these are filled. In the case of lexically-specified phrase boundaries, such as those occurring between the conjuncts of recursive and, incremental string-building of this kind has the advantage that once a phrase has been completed, it can be pushed onto the (local) linguistic history. In principle, to the extent that the analysis tree is left-headed, words can be output incrementally. This is not done in the current implementation.

5.4.4 Re-using material from the linguistic history

In this section I give more detail of the implementation of the ideas presented in Section 4.2.4, where it was proposed that a limited recall of surface analysis trees could be of benefit both to the speaker, in reducing lexical search and structure-building operations, and to the hearer, who is thereby absolved from some of the effort of parsing. I describe first how the linguistic history is kept up-to-date and
provided as a knowledge source to the linguistic generation algorithm. I then give the algorithm used for surface reuse, and suggest how instructions arising out of accessibility history reuse during description graph generation may focus the search in surface reuse.

5.4.4.1 Maintaining and accessing the linguistic history

As a result of linguistic generation, and after parsing, the current analysis tree is pushed onto a list representing the linguistic history. In the case of parsed input, this happens once the interpretation process has returned bindings for variable discourse entity identifiers. The first \( n \) entries, where \( n \) is the current length of the accessibility history, are provided when the procedure \texttt{generate} is called. The linguistic history is accessed as an environment variable, which is ‘threaded’ through the Prolog procedures for \texttt{generate}, \texttt{gen_from_entry}, and any others that need to access it. At the start of generation, the list of previous analysis trees is read in. During generation if a lexically-marked phrase is completed, this is pushed onto the front of the linguistic history. For example, the entry for \texttt{and ((A\backslash A)A} in categorial grammar) is represented in UCG as (5.31):

\[
\begin{align*}
\text{head} : & \text{Head} \\
\text{syn} : & \left[ \begin{array}{c}
\text{head} : \text{Head} \\
\text{sem} : E
\end{array} \right] \\
\text{args} : & \left[ \begin{array}{c}
\text{dir} : \text{post} \\
\text{adj} : \text{next} \\
\text{ann} : \text{li}
\end{array} \right] \\
\text{order} : & \left[ \begin{array}{c}
\text{dir} : \text{post} \\
\text{adj} : \text{next}
\end{array} \right] \\
\text{type} & \text{and}
\end{align*}
\]

(5.31)

The feature value \( \text{ann} : \text{li} \) on the first argument indicates that an obligatory phrase boundary should follow it. This is intended to implement the phrase boundaries, or ‘intonational commas’, that separate the items of lists.\(^4\) The \texttt{generate} algorithm detects when a constituent so delimited is complete, and places it on the front of the linguistic history, where it is available for \texttt{gen_from_entry} during subsequent stages of the present generation cycle. However the entry saved on the

\(^4\)Apart from this example of how phrasing that is arguably syntactic in origin may be lexically specified, I assume in general that phrasing is assigned outside the linguistic generation component.
permanent linguistic history is not divided into phrases.

5.4.4.2 The reuse algorithm

The algorithm implemented is essentially the same as that discussed in Section 4.2.4, page 134. Firstly, it is necessary to find suitable candidate analysis trees or subtrees in the linguistic history. This is done by searching previous entries in order of recency, considering first an analysis tree, then its subtrees. To avoid unnecessary search while matching is attempted, a heuristic is applied to find likely candidates. This requires that the target semantics, $S_{trg}$ and that of a candidate entry: $A_{old} : <sem>$ be sufficiently close that the most specific semantic type which subsumes both their semantic types should be a lexical type: ie, one for which lexical entries exist. For example, referring to the semantic hierarchy shown in Figure 4.5, TRAVEL is the lowest node in the hierarchy which subsumes both FLIGHT and DRIVE, for which there are lexical entries.

Once a candidate has been proposed, the reuse algorithm may be attempted. This is specified in (5.32). The relation MATCH, which is used to compare the semantics of old and new structures, is partitioned into: $MATCH_{=}$ (equality matching), $MATCH_{lex}$ (lexical matching), and $MATCH_{sc}$ (semantically close lexical matching). The two latter combine some lexical search with matching, so can be treated as functions which return uninstantiated lexical entries—ie, ones which have yet to combine with other constituents—in the case of $MATCH_{lex}$, one that is common to both structures; in the case of $MATCH_{sc}$, a lexical entry which is appropriate for $S_{trg}$ and semantically close to the entry for $A_{old}$.

The function $GEN\_FROM\_ENTRY$ takes as input a target semantics $S_{trg}$, and a previous analysis (sub)tree $A_{old}$, where $MATCH(S_{trg}, A_{old} : <sem>)$, and builds an analysis tree $A_{trg}$ as follows:
\textbf{gen_from_entry}(S_{\text{trg}}, A_{\text{old}}):$

\begin{align*}
  \text{if } \text{match}(S_{\text{trg}}, A_{\text{old}}; <\text{sem}>) \text{ then} \\
  & A_{\text{trg}} = A_{\text{old}} \\
  \text{else if } Lex_{\text{old}} = \text{match}_{\text{lex}}(S_{\text{trg}}, A_{\text{old}}; <\text{sem}>) \text{ then} \\
  & \theta = \text{unify}(S_{\text{trg}}, Lex_{\text{old}}; <\text{sem}>) \\
  & \text{Args} = \text{apply}(\theta, Lex_{\text{old}}; <\text{args}>) \\
  \text{foreach} (sub_0, sub_{\text{old}}, sub'_0) \in \text{Args} \times A_{\text{old}}; <\text{subs}> \times \text{Args}' \\
  & sub'_0 = \text{gen_from_entry}(sub_0; <\text{sem}, sub_{\text{old}}>) \\
  & A_{\text{trg}}; <\text{lex}> = Lex_{\text{old}} \\
  & A_{\text{trg}}; <\text{args}> = \text{Args}' \\
  \text{else if } Lex_{\text{sc}} = \text{match}_{\text{sc}}(S_{\text{trg}}, A_{\text{old}}; <\text{sem}>) \text{ then} \\
  & \theta = \text{unify}(S_{\text{trg}}, Lex_{\text{sc}}; <\text{sem}>) \\
  & \text{Args} = \text{apply}(\theta, Lex_{\text{sc}}; <\text{args}>) \\
  \text{foreach} (sub_0, sub_{\text{old}}, sub'_0) \in \text{Args} \times A_{\text{old}}; <\text{subs}> \times \text{Args}' \\
  & sub'_0 = \text{gen_from_entry}(sub_0; <\text{sem}, sub_{\text{old}}>) \\
  & A_{\text{trg}}; <\text{lex}> = Lex_{\text{sc}} \\
  & A_{\text{trg}}; <\text{args}> = \text{Args}' \\
  \text{else generate}(S_{\text{trg}}) \\
  \text{endif}
\end{align*}

(5.32)

the first three branches of the conditional correspond to the cases \text{match}_=, \text{match}_{\text{lex}}

and \text{match}_{\text{sc}} respectively. \text{unify}, ordinary unification of feature structures, is
defined to return a substitution \(\theta\); this may then be applied to some other feature
structure using \text{apply}, if the effect of exporting the unification to its environment
is required.

\text{gen_from_entry} is combined with the basic algorithm \text{generate} by testing
for re-use before trying ordinary lexical access. In the implementation, it is necessary
that the default call to \text{generate} (last branch of the conditional in (5.32)) is only
allowed in embedded calls. Otherwise cyclic recursion may result. If the Prolog
call to \text{gen_from_entry} fails at the top level because \text{match}_=, \text{match}_{\text{lex}} and
\text{match}_{\text{sc}} all failed, then other candidates may be tried; if none of these are good,
then \text{generate} with ordinary lexical access is used.

5.4.4.3 Combining reuse instructions from the accessibility history

The use of the accessibility history in building description graphs (cf. Section 5.3.2)
suggests that information from that stage can be put to use in proposing candidates
for surface re-use. Indeed it would be surprising if this were not the case. If the buffer
sizes are the same, the entries in the accessibility history are necessarily contained in
the semantic components of the corresponding linguistic history entries. Moreover,
the relation $\prec_{DM}$ defined over discourse entities is expected to have an effect on the
prominence ordering defined at the surface level.

As seen in Section 5.3.3, the description graph procedures produce instructions
for use at later stages. Such instructions point to where surface-reuse may be at­
temted, what suitable antecedents are, and what form of matching is appropriate.

The instructions are used as follows: when \texttt{generate} is building analysis tree
$A$, and $A:\langle \text{root sem id} \rangle$ is a node $e$ in $ES_{DG}$ for which an instruction is defined, then:

1. if $Instruction_{DG}(e) = \langle \text{new} \rangle$ then \texttt{generate} with no reuse;

2. if $Instruction_{DG}(e) = \langle i, eq \rangle$ for some $i \in \text{EntryNos}$, then find the subgraph
   of $LH_i$ whose semantic root is $e$, and copy this to $A$;

3. if $Instruction_{DG}(e) = \langle i, mod \rangle$ for some $i \in \text{EntryNos}$, then run \texttt{generate}
   with antecedent the subgraph of $LH_i$ whose root is $e$. However, when at some
   recursive stage any semantic node $w$ is reached such that $Instruction_{DG}(w) = \langle i, neq \rangle$ or $Instruction_{DG}(w) = \langle \text{new} \rangle$ then run \texttt{generate} with no reuse;

4. if $Instruction_{DG}(e) = \langle i, par \rangle$ for some $i \in \text{EntryNos}$ then do as for a \texttt{mod}
   instruction, except that MATCH$_{\varepsilon}$ may be used as an alternative to MATCH$_{lex}$.

Instructions for reuse and modification derived from the accessibility history
reduce the need to search for candidates for surface reuse. However they do not nec­
essarily find all such candidates. Distinctions may be made at the surface level which
are not available within the accessibility history. Although the lexicon encourages
the semantics of numeric expressions to be compositional, in the implementation
these may be stored in the discourse model as integers, with conversion taking place
at the linguistic interface and prior to linguistic generation. This means that the
potential for reuse between the integers 37 and 27 is not detected until the surface
reuse stage. On the other hand, if the semantics of numbers is compositional at
the description graph stage, then commonality between 27 and 17 will be detected.
But `gen_from_entry` will fail for 17 with 27 as the antecedent structure, because unlike 27, the lexical entry for 17 has no substructure. It will also fail for 27 following 17; however this time the prominence ordering over the discourse entities corresponding to the ten and the units can be realised on the surface form `twenty >seven<`. In Appendix D (pages 339–343) an example is given of the generation of `eight twenty seven`, following on from `eight seventeen`. In order to facilitate matching, in the lexicon `seventeen` is 'morphologically' decomposed into `seven` and `teen`.

5.4.4.4 Prominence ordering resulting from surface reuse

Apart from the prominence characteristics of discourse entities recorded in the information profile, at the level of linguistic generation prominence assignment to surface structures may come about in one of two ways, either because it is lexically specified, or as a side-effect of the operation of the surface-reuse algorithm. Lexically specified prominence is in the current implementation restricted to a prominence order at a local level, for example between elements of a noun compound. These are defined by a feature on relevant lexical entries that determines the relative prominence ordering of head and arguments.

Prominence as the result of surface reuse may be defined analogously to the accessibility ordering over discourse entities $\prec_h$. However unlike nodes in the description graph, nodes in the surface analysis tree are not identified by unique symbols, but rather by their position in the tree. It is therefore necessary to attach features specifying prominence to the analysis tree itself. Moreover, local configurations in the tree are not represented in the usual constituency manner, since the node representing a lexical head is absorbed into the non-terminal that dominates the phrase containing the head and its arguments, as shown in Figure 5.12. In (a), corresponding to conventional constituent structure, `Head` is a daughter of the non-terminal `Head'`. In (b) on the other hand, `Head` occupies the place of the non-terminal; its contribution to the surface string is via its feature `lexeme`, with word order being dictated by features on `Arg1 ... Arg3`. The alternative (b), which I use, collapses the nodes `Head` and `Head'` into a single node. This is possible because `Head` and `Head'` do not disagree about feature assignments, but only differ in their degree of
Figure 5.12: Heads as terminals and nonterminals: comparison of representations instantiation. This representation is adopted for reasons of economy; the features of the lexical head without the arguments can if needed be recaptured by ignoring the contribution due to \{Arg_1 \ldots \} in Head. This leaves the problem however of distinguishing between the focal properties of the lexical head and the phrasal node representing it. For example, if the reuse instruction \( (n, eq) \) is assigned on Head in Figure 5.12 (b), it is not clear how much of the surface structure this applies to. The distinction can be made using the feature \textit{scope}, with two values: \textit{lexical} and \textit{phrasal}. For \textit{eq} instructions, this is \textit{phrasal}; the default is \textit{lexical}.

If a call to \texttt{gen\_from\_entry} guided by an instruction \( RI \) succeeds, \( RI \) is copied onto the root node of the (sub-) analysis tree built. If recycling of a previous (sub-) analysis tree takes place where no reuse instruction was present (generally because of a difference in granularity between surface semantics and discourse-model semantics), then an appropriate reuse instruction is written directly onto the relevant surface node.
5.5 Generating prosodic information

In Section 5.3.3 I related how a feature assignment on discourse entities denoting intentional focus was obtained and how a partial ordering denoting accessibility was arrived at. Section 5.4.4.4 described the derivation of a featural assignment on nodes of the surface analysis tree, as a result of surface reuse. These components of focal assignment are shown together in Figure 5.13. Also shown are the reuse instructions which form part of the input to the surface-reuse component. It will be observed that whereas the domain of intentional focal assignment and that based on accessibility is that of semantic objects, ie. nodes in a description graph, for prosodic realisation focal marking must be on surface structure. In Section 5.5.1 I show how features on surface nodes are derived from those on semantic nodes, and discuss how conflicts with features assigned during linguistic generation may be resolved.

I then turn to prosodic realisation in the Sundial system. Section 5.5.3 describes how information regarding prosody is communicated to the synthesis-by-rule component. Finally, Section 5.6 gives details of current work regarding intonation contours.
5.5.1 Combining prominence information from the discourse and surface levels

The pattern of prominence derived on the surface structure as a result of surface recycling is close to being directly interpretable as instructions for accent placement. This is not so in the case of the markings on discourse elements. The problem arises of which node in a derivation tree produced by a categorial grammar should be the one to which focal information is attached. If there is a one-one correspondence between semantic entities and lexical entries, then the information may be attached to the surface node at which the lexical entry is introduced. In the analysis trees used in the Sundial generator, in which non-terminals are allowed to stand for heads (cf. Figure 5.12) this means that any focal properties associated with a discourse entity corresponding to such a head must be restricted to lexical scope. Unfortunately, the mapping between discourse entities and surface nodes need not be so straightforward. Examples (5.33–5.34) illustrate cases of many-one and one-many correspondences.

(5.33) fly

\[
\begin{array}{c}
\text{type: journey}
\
\text{theinstrument: plane}
\end{array}
\]

(5.34) carry out a payment

\[
\begin{array}{c}
\text{type: pay}
\end{array}
\]

SIL distinguishes between modes of travel using the theinstrument slot (5.33). While this is no more general than the domain of plane and train journeys, it has the advantage of explicitly introducing the existentially quantified mode of transport as a discourse entity. It means however that the single lexical item fly has more than one discourse entity associated with it. Conversely, in an alternative domain dealing with financial transactions, the primitive PAY can be linked to the phrase carry out a payment. This choice over the simple verb pay might for example be made when no object for transitive pay was available, or when a paraphrase was required.

In order to formulate rules which will handle these cases, I distinguish between discourse entities introduced and not introduced by a lexical entry. The first category consists of those discourse entities which become associated with a lexical head as a result of unification of its semantics with the input, before any subcategorial argu-
ments or any modifiers have been generated. The second category is the complement of the first: ie, those entities in the input which are not associated specifically with the head and therefore must be associated with arguments or modifiers. The rules for associating discourse focal properties with surface forms are therefore:

1. Copy any introduced focal properties onto the lexical head, at a stage in the \textsc{generate} algorithm (cf. 5.29) before the recursive generation or raising takes place.

2. The textual scope of such properties may extend beyond the lexical head proper, to cover any surface nodes dominated by a projection which doesn't introduce new semantic material.

Rule 2 might be viewed as reintroducing the notion of phrasal focal domains. However it should be observed that it only operates on domains whose semantics is defined entirely lexically, which may be considered to be frozen expressions, or expressions built out of words some of which are meaningless placeholders.

5.5.2 Generating the focus ordering

In Section 4.3.2 I showed how semantic focus, viewed as a partial ordering over discourse entities, can be derived from partionings based on (historical) accessibility, inferrability, informativeness, and intention. These properties of discourse entities are available when the description graph is built; it would therefore be possible to derive the ordering at that time. Delaying it until after the surface structure has been built, however, has the advantage that reuse instructions which cannot be used in the generation process are dropped, while new ones may be formulated as the result of the comparison of surface constituents. Moreover only discourse entities that correspond to surface constituents need to be considered. The main overhead is that featural properties of the description graph, such as intentional focus, have to be retained for use by processes that do not have direct access to Dialogue Manager knowledge bases.

The ordering algorithm takes the set of semantic identifiers attached to nodes of the surface tree: \textit{SurfaceIds}, and considers successive partionings based on featural
properties passed down in the environment. Priority is given to earlier partitionings.

In the definition of the algorithm given in Figure 5.14, the notation $\overline{\text{SET}}$ is used to describe the complement of set $\text{SET}$ with respect to $\text{SurfaceIds}$. The definition of $\text{FOCUS\_ORDER}$ is essentially procedural, each successive operation of $\text{ADD\_P\_ORD}$ operating on the output of the previous one.

Partial orderings are originally specified as partitionings, in the form

$$\text{SubSet}_a \prec \text{SubSet}_b \prec \text{SubSet}_c \prec \ldots$$

where the subsets are disjoint partitions of all or part of $\text{SurfaceIds}$. Internally the partial ordering is represented as a set of ordered pairs $\{e_i \prec e_j \mid e_i, e_j \in \text{SurfaceIds}\}$.

The function $\text{ADD\_P\_ORD}$ takes a partition, and a canonical ordering (which may be empty), and returns a new canonical ordering which combines the two, except in cases of conflict where priority is given to the original ordering. Pairs which are derivable from the transitivity of $\prec$ are removed.

The relations $\text{HAS\_IFOCUS}$, $\text{REUSE}$, $\text{PRONOUN}$, $\text{DEICTIC}$, $\text{BM\_ID}$ are used to
partition SurfaceIds according to their properties recorded in the environment. 

one_to_many_ord is different in that pairs from SurfaceIds are examined, to find out which if any are structurally in the relation one_to_many within the semantics. For example, if city1 corresponds to london, and airport1 to heathrow, then the rule (5.24) in Section 5.3.3 will result in city1 < airport1 being added to the ordering at this stage.

The partial ordering derived by focus_order could be mapped directly onto a relational prosodic representation, of the kind proposed by Ladd (1986, 1988). For the purposes of this implementation however, it is necessary instead to derive emphasis levels. The procedure totalise_order takes the set of ordered pairs produced by focus_order and works up from the lowest elements of the ordering graph, assigning successively higher integers to members of SurfaceIds. The initial value (for the lowest members) is zero or one, depending on whether or not there are ids with reuse(Mod, Id, Env). The ordering in (5.35) for example results in the levels assignment shown in (5.36):

\[(5.35) \quad \{a < b, b < c, c < d, b < e\}\]
\[(5.36) \quad \{a : 1, b : 2, c : 3, e : 3, d : 4\}\]

In the implementation, the possibility that prominence numbers may become large, if a large number of elements are ordered, is avoided by banding them. An attractive alternative would be to map the partial ordering directly onto a relational prosodic structure.

The resulting total ordering is not in general equivalent to the partial ordering from which it is derived. In Section 5.5.3 I explore how this loss of information may be exploited in a choice of realisation possibilities.

5.5.3 Producing the annotated text

The interface between the generation components, and the synthesis-by-rule component takes the form of an annotated string. The string contains the text of the utterance in orthographic form. This is punctuated by a number of markers, indi-
cating attributes of the following kinds:

1. Parts of speech;

2. Major phrasal boundaries;

3. Constituents with idiosyncratic stress properties, such as noun compounds, and time expressions;

4. Explicit phrasal boundaries;

5. Focal domains;

6. Dialogue act labels, or other indications of contour.

Because of the linear nature of representations within the rule-based component, and the need for consistency with representations used purely for text-to-speech, the grammatical symbols used are not those of UCG.

During generation, the field _phon_of a sign contains not a string, but a complex feature structure reflecting its derivation according to the rules of functional application. In addition there may be (surface) node identifiers, and phrasal or lexical category labels required as annotations when the string is passed to the rule-based synthesis component. Once _generate_ has succeeded for the whole utterance, the _phon_ component of the root sign is traversed by the procedure _LINEARISE-PHON_, which produces the annotated surface string. The annotations include focus markings (described in more detail below). Symbols corresponding to focal levels—with special symbols for emphasis or rhetorical focus—are looked up, and inserted in the string either to bracket or prefix the constituent in question. Such marking normally applies to the lexical head alone; that is, unless _pfocus : scope_ is marked as phrasal, in which case it will extend to arguments, so long as they themselves do not have incompatible _pfocus_ values.

Details of the interface notation are given in House and Youd (1990); here I concentrate on focal domain labelling.

Focal domains are annotated with pairs of bracketing symbols, as summarised in (5.37).
Additionally, the prefixes '5' and '6' before individual words may be used to achieve prominence which is greater than the default, but which falls short of emphasis. These symbols are assigned according to the table in (5.38). In cases where a word prefix is used instead of a pair of bracketing symbols, " is used to indicate an empty closing bracket.

\[
\begin{align*}
\text{bracketing_sym}(\text{caller\_authd}, e:, :e). \\
\text{bracketing_sym}(\text{system\_authd}, e:, :e). \\
\text{bracketing_sym}(\text{system\_relaxed}, e:, :e). \\
\text{bracketing_sym}(\text{modf\_view}, e:, :e). \\
\text{bracketing_sym}(\text{rf}, r:, :r). \\
\text{bracketing_sym}(5, 6,"). \\
\text{bracketing_sym}(4, 6,"). \\
\text{bracketing_sym}(3, 6,"). \\
\text{bracketing_sym}(2, 5,"). \\
\text{bracketing_sym}(1, +:, :+). \\
\text{bracketing_sym}(0, |:, :|).
\end{align*}
\]

The first four of these base the bracketing decision on intentional focus. The intentional focus marking system\_relaxed refers to the case where the system has made an unauthorised modification as a result of relaxation of the caller's original constraints. The value rf refers to the theme of a rhetorical-dependency pair (cf. Section 5.3.3). In each of these cases, the result is a bracketing pair, which is used to delimit the constituent. Otherwise, the annotation is chosen on the basis of the integer assigned by totalise\_order (Section 5.5.2). If this is zero, it is mapped onto a pair of bracketing symbols indicating deaccenting; if 1, it becomes the pair +: ... :+, indicating normal (default) level of focus; because the prosody generation component does not require annotations in this case, these are subsequently deleted. Otherwise it is mapped onto level prefixes '5' or '6'.

The realisation of the prominence order in terms of levels of accent heights is somewhat arbitrary. As noted, information is lost when a partial ordering is converted into a total one. The speaker has the option, for example, of using more accents and distinguishing among these according to accent height, or fewer accents, in which case the accented/deaccented distinction becomes important. In Bolinger's
terms, the choice of more accents or fewer may be accounted for in terms of intended
'power'. Such a decision may be traced to the amount of insistence required; for
example, a repetition would warrant more accents. Such pragmatically-based deci-
sions are not made in the current implementation. Instead, it is possible to set a
global parameter which determines the power level of focus annotations. Currently
there are three options: the default—described in (5.38), 'low'—where integer values
of the totalised order are moved down by one before the symbols are looked up, and
'high', where they are moved up.

As an example, consider the string bee ay one two three, with emphasis on one.
The surface features are as follows:

```
h : h : h : h : morform : bee
   lexical : n
   phrasal : np
   pfocus : semid : carrier1
     ifocus : dg_ifocus
     reuse : mod
     scope : lexical
a : morform : ay
   lexical : n
   pfocus : semid : gi(string,a)
   scope : lexical
   reuse : eq
   ifocus : dg_ifocus
a : morform : one
   lexical : n
   pfocus : reuse : neq
   scope : lexical
   ifocus : system_authd
   semid : gi(digit,1)
a : morform : two
   lexical : n
   pfocus : semid : gi(digit,2)
   reuse : mod
   scope : phrasal
   ifocus : dg_ifocus
a : morform : three
   lexical : n
   pfocus : semid : gi(digit,3)
   reuse : mod
   scope : lexical
   ifocus : dg_ifocus
```

The markings generated by the three level settings are (starting with the low-
est):

(5.39)  
```
# :#bee#n: || :#ay#n: |e :#one#n:e |#two#n: || :#three#n: |np#.

bee#n#ay#ne : #one#n:e #two#n: three#nnp#.
```

246
In all cases, the emphatic marker results in *one* receiving enhanced prominence. Figure 5.15 shows the $F_0$ graphs for the three strings of (5.39). The prosody generation component is part of a text-to-speech synthesis system, taking as input an annotated string, and producing synthesized speech. Its description lies outside the scope of this thesis (see however House 1990, House and Youd 1990).
5.6 Generating intonation contours

The implementation described in this chapter is primarily a computational account of the production of prosodic prominence. The lack of explanatory power of the results of Chapter 3 has led me to consider contour generation separately. However, as I observed in Section 3.5, even a probabilistic model provides a better basis for contour decisions than punctuation-based heuristics. This section presents such a model, which selects symbolic descriptions of contours capable of being combined with the prominence patterns discussed elsewhere in this chapter.

The analysis presented in Section 3.4 was compiled into a set of Prolog relations, of the form

\[\text{ctr}(\text{DACT}, \text{Speaker}, \text{Head}, \text{Nucleus}).\]

\textit{Speaker} may be any one of the four speakers; alternatively results can be pooled for all speakers. Where there is inadequate information in the corpus, default rules have been used. Below are the rules for speaker JQ:

\[(5.40)\]

\[\text{ctr([repeated], jq, [upst:4, level:3, downst:2], [hl:7, hlh:2]).}\]
\[\text{ctr([init, confirm, open], jq, [level:5, upst:5], [hl:10]).}\]
\[\text{ctr([init, confirm, -strong], jq, [level:3, mixed:3, upst:3], [hl:6, hlh:3]).}\]
\[\text{ctr([init, confirm], jq, [downst:4, level:3, upst:3], [hl:6, hlh:4]).}\]
\[\text{ctr([init, correct], jq, [upst:10], [hlh:10]).}\]
\[\text{ctr([init, query, default], jq, [level:5, upst:5], [hlh:10]).}\]
\[\text{ctr([init, query, open], jq, [level:5, upst:2, mixed:1, downst:1], [hlh:8, hu:1]).}\]
\[\text{ctr([init, query, value], jq, [level:5, upst:10], [hl:10]).}\]
\[\text{ctr([init, alta], jq, [mixed:3, upst:3, downst:3], [hl:10]).}\]
\[\text{ctr([init, preclose], dft_spkr, [level], [lhm]).}\]
\[\text{ctr([init, opening], dft_spkr, [level], [lhm, lhm]).}\]
\[\text{ctr([init, repeat], jq, [], []).}\]
\[\text{ctr([init], jq, [upst:3, level:3, downst:2, mixed:1], [hlh:4, hl:4]).}\]
\[\text{ctr([resp, topic], jq, [downst:5, mixed:2, upst:2], [hl:10]).}\]
\[\text{ctr([resp], jq, [level:5, mixed:1, upst:1, downst:1], [hl:9]).}\]
\[\text{ctr([preinit, -alta2], jq, [level:5, upst:3, mixed:1], [hlh:5, hu:1, lhm:1, hl:1]).}\]
\[\text{ctr([mod], jq, [upst:5, level:2, downst:2], [hl:5, hlh:6]).}\]

A contour is assigned using the normal Prolog order of search, finding the most specific rule compatible with the given dialogue act. In many cases a speaker used more than one contour for a given dialogue act description; numbers reflect relative frequency, and are used to guide the random choice of contour description for a given utterance. So for example, the dialogue act description \textit{(init; query; open)}
will probably result in the combination level – hlh being selected. Once chosen, a contour description may be converted to annotation symbols, to be sent to the rule-based synthesis component.

5.7 Producing an utterance: an example

This section presents in some detail an example of the description and generation mechanisms described in Sections 5.3 and 5.4. In order to clarify the use of histories it is useful to provide all the relevant preceding context, even when describing the generation of a single utterance. I therefore give the entire dialogue, which is an extension of the one given in (4.47):

\[
\begin{align*}
C1: & \quad \text{i want to travel from London to Paris} & \text{init; task; topic} \\
A1: & \quad \text{what time do you want to arrive in Paris} & \text{init; query; open} \\
C2: & \quad \text{two thirty} & \text{resp; query; open} \\
A2: & \quad \text{two thirty} & \text{init; confirm; \textit{strong}} \\
A3: & \quad \text{what time do you want to leave London} & \text{init; query; open} \\
C3: & \quad \text{leave London at twelve thirty} & \text{resp; query; open} \\
\end{align*}
\]

In this section, I describe how the system generates the utterance A3.

5.7.1 Description generation

As input the Belief Module receives the description instruction from the Message Planner:

\[
\text{msg}(bm, mp, \text{request\_description\_object\_in\_ctx}( \\
(5.42) \quad mp5, \\
\quad \text{know}(\text{shared}, \text{avs}(dbflight1, [\text{sourcetime : A, sourcecity : london}], B))).)
\]

Essentially this requires the description of the task-oriented input: \([id : dbflight1, type : dbflight, sourcetime : A, sourcecity : london]\). The instruction suffix \(\ldots\_\text{in\_ctx}\) requires a lambda-abstracted description, with \(A\) the abstracted variable.

Figure 5.16 shows a graph representing the current state of the belief module, in the current view. For simplicity of representation, the graph is shown as a tree,
Figure 5.16: State of Belief Module prior to the generation of $A_3$
with the node `single_journey`1 as root. Re-entrant subgraphs (cf. Figure 4.1) are identified using the `id` feature.

Before the description instruction is issued, all new material required by the description (such as the subgraph whose root is `s_time2`) has been added. Firstly, the set of core identifiers is found. These refer to objects which are inferentially connected to the task-oriented material in the description instruction. If possible, the current view is used; if however values at the current view conflict with those in the description instruction, a closely connected view is used. In the present case there is no conflict of views, and the identifiers `{minutes2, s_time2, hour2, city2, location2}` are returned.

The accessibility history is as follows:

1: [id : `s_time1`, type : `s_time`, ...] (A2)
2: `backref`(1, [ ]) (C2)
3: [id : `s_time1`, type : `s_time`, thedesc : ...] (A1)
4: [id : `go1`, type : `go`, ...] (C1)

`describe_ids` finds the core graph with root `location2` (in entry 4) and builds a new graph corresponding to the root `s_time2`. Next the procedure `find_spanning` attempts to discover or create a graph which will span both [id : `location2`, ...] and [id : `s_time2`, ...]. Since an abstracted structure is required, the search begins first with previous abstracted structures; in this case, 3 : [id : `s_time1`, type : `s_time`, thedesc : [id : `arrive1`, ...]]. Starting with the normal form of this (root `arrive1`), the procedure `tandem_possible` attempts to find if this is possibly structurally parallel to the target description graph: ie, it contains subgraphs which match some of the core graphs. `tandem_possible` succeeds, returning the following path and instruction information:

(5.43) `location1` [theplace] {city2 : ri(neq), location2 : ri(par)}
`s_time1` [thetime, thetime] {s_time2 : ri(typeq)}

The information in (5.43) gives the paths in the target graph at which `location1` and `s_time1` are to be found, and also indicates the appropriate reuse instructions: ri(neq), ri(par) and ri(typeq). The labels neq and par come about because the subgraphs with roots `location1` and `location2` are structurally parallel, differing at the level of `city1` and `city2`. The reuse label `typeq` is a variant of eq,
used when the antecedent entity is of the same type but not identical, but where both have been used to represent unknown entities. **FIND_SPANNING** now proceeds with the core graphs, and the paths found as input. For the first core graph (root *location2*), the path [*theplace*] is followed in reverse, leading to the candidate root *depart1*. For the next core graph, it is verified that following the (reversed) path [*thetime, thetime*] leads again to this root node. The lexical constraint on description-graph nodes of type *depart* is now checked; still lacking is a subgraph at the path < *thejourney* *thepassenger* *theindividual* >. This is now added. Finally, a lambda abstract (with respect to the unknown material) is produced (5.44).

![Diagram](image)

Note that the path: < *thejourney* *thepassenger* *theindividual* > is aliased to < *thetheme* >, which is required by the surface semantics in the lexicon. The reuse instructions added are as follows:

- *city2* (3, neq)
- *location2* (3, par)
- *s_time2* (3, typeq)
- *depart1* (3, par)
- *caller* (3, eq)

The first three of these are the ones proposed by **TANDEM_POSSIBLE**; the instruction for *depart1* is because the roots of the parallel structures (*arrive1* and *depart1*) are different; the one for *caller*, although not proposed by **TANDEM_POSSIBLE**, is found during graph comparison in **FIND_SPANNING**.

252
5.7.2 Linguistic generation

The input to linguistic generation is the graph:

In addition to this, and the linguistic history, the information profile describes focal features of discourse entities, as derived at the planning stage.

s_time2 [ifocus: dg_ifocus, owner: inferred(system), reuse: [ri(3, typeq)]
depart1 [owner: inferred(caller), reuse: [ri(3, par), new]]
time_point2 [owner: inferred(system), reuse: [new]]
location2 [ifocus: dg_ifocus, owner: caller, reuse: [ri(3, par)]
city2 [ifocus: dg_ifocus, owner: caller, reuse: [ri(3, neq)]
caller [owner: caller, reuse: [ri(3, eq)]
discourserole2 [owner: caller, reuse: []]

In the following, I show the recursive working of the procedures generate and gen_from_entry. Inputs are identified by the syntactic head category and semantic type; in the case of outputs an analysis tree is shown, in greatly reduced form.
At the outermost call to \texttt{generate}, the syntactic head category is not known:

\begin{verbatim}
call(0) : GENERATE
  [] : - : S_TIME

  call(0) : GEN_FROM_ENTRY
  [] : n : S_TIME
  INSTRUCTION : ri(1, noinst)
  ENTRY : 
  twothirty : n : S_TIME
    two : n : S_TIME
    thirty : n : MINUTES →

  fail(0) : GEN_FROM_ENTRY...
\end{verbatim}

In the implementation level the lexeme \textit{what} is treated as a kind of raiser. It is not currently possible to exploit the reuse instruction associated with \texttt{s_time2}. The failure of \texttt{GEN_FROM_ENTRY} occurs when a match with the more recent \textit{two thirty} is attempted.

\begin{verbatim}
call(1) : GENERATE
  [] : n : S_TIME →
    [] : n : S_TIME
    [] : det : UNSPEC

  done(1) : GENERATE
  time : n : S_TIME
    time : n : S_TIME

  The entry \textit{what} is treated as (categorial) \((v[wh]/v[inv])/n\)

  call(1) : GENERATE
  [] : v : WANT →
\end{verbatim}

In the information profile a reference to \textit{want1} is missing. However comparison (on the basis of \textit{<syn head>} and \textit{<sem type>}) with the subtree from entry [3] produces a match, so that \texttt{GEN_FROM_ENTRY} can be called:

254
call(1): \texttt{GEN\_FROM\_ENTRY}
\[ \texttt{\vdash v: WANT} \rightarrow \]
\texttt{INSTRUCTION: ri(3, noinst)}
\texttt{ENTRY:}
\begin{align*}
do.you.want.to.arrive.in.paris.at & : \texttt{v: WANT \rightarrow} \\
do & : \texttt{v: WANT} \\
you & : \texttt{n: INDIVIDUAL} \rightarrow \\
want.to.arrive.in.paris.at & : \texttt{v: WANT \rightarrow} \\
want & : \texttt{v: WANT} \\
to & : \texttt{v: UNSPEC} \rightarrow \\
arrive.in.paris.at & : \texttt{v: ARRIVE \rightarrow} \\
arrive & : \texttt{v: ARRIVE} \\
in.paris & : \texttt{prep: LOCATION} \rightarrow \\
in & : \texttt{prep: LOCATION} \\
paris & : \texttt{n: LOCATION} \rightarrow \\
paris & : \texttt{n: LOCATION} \\
at & : \texttt{prep: S\_TIME} \rightarrow \\
at & : \texttt{prep: S\_TIME} \\
\end{align*}

\begin{align*}
call(101) & : \texttt{GENERATE} \ldots \\
call(101): \texttt{GEN\_FROM\_ENTRY} \\
\vdash v: \texttt{WANT} \rightarrow \\
\vdash v: \texttt{WANT} \\
\vdash n: \texttt{INDIVIDUAL} \\
\texttt{INSTRUCTION: ri(3, noinst)} \\
\texttt{ENTRY:}
\begin{align*}
want.to.arrive.in.paris.at & : \texttt{v: WANT \rightarrow} \\
want & : \texttt{v: WANT} \\
\texttt{o} & : \texttt{n: INDIVIDUAL} \rightarrow \\
\end{align*}
\end{align*}

At this stage in the recursion, one of the original reuse instructions is reached. Generation continues to be with respect to entry [3]:

\begin{align*}
call(201): \texttt{GEN\_FROM\_ENTRY} \\
\vdash v: \texttt{DEPART} \rightarrow \\
\vdash v: \texttt{DEPART} \\
\vdash n: \texttt{INDIVIDUAL} \rightarrow \\
\texttt{INSTRUCTION: ri(3, par)} \\
\texttt{ENTRY:}
\begin{align*}
arrive.in.paris.at & : \texttt{v: ARRIVE \rightarrow} \\
\end{align*}
\end{align*}

\begin{align*}
call(301): \texttt{GEN\_FROM\_ENTRY} \\
\vdash n: \texttt{LOCATION} \rightarrow \\
\texttt{INSTRUCTION: ri(3, par)} \\
\texttt{ENTRY:}
\begin{align*}
in.paris & : \texttt{prep: LOCATION} \rightarrow \\
\end{align*}
\end{align*}

Structural parallelism works despite the difference in case-markers: (in \textit{paris} vs \textit{\underline{a}} \textit{london}. This is because, where matching of raised expressions fails, the corresponding unraised ones are tried. Similarly, matching does not depend on morphological
instantiation.

\[ \text{exit}(301) : \text{GEN\_FROM\_ENTRY} \]
\[ \text{london} : \text{n} : \text{LOCATION} \]
\[ \text{london} : \text{n} : \text{LOCATION} \]

\[ \text{call}(301) : \text{GEN\_FROM\_ENTRY} \]
\[ [] : \text{prep} : \text{S\_TIME} \rightarrow \]
\[ \text{INSTRUCTION} : \text{rt}(3, \text{typeq}) \]
\[ \text{ENTRY} : \]
\[ \text{at} : \text{prep} : \text{S\_TIME} \rightarrow \]
\[ \text{at} : \text{prep} : \text{S\_TIME} \]

\[ \text{exit}(301) : \text{GEN\_FROM\_ENTRY} \]
\[ \text{at} : \text{prep} : \text{S\_TIME} \rightarrow \]
\[ \text{at} : \text{prep} : \text{S\_TIME} \]

Once the arguments of \textit{leave} have been recursively generated, the sub-analysis tree corresponding to the inner \( v[bse]/n \) is complete.

\[ \text{exit}(201) : \text{GEN\_FROM\_ENTRY} \]
\[ \text{leave} \text{.london.at} : v : \text{DEPART} \]
\[ \text{leave} : v : \text{DEPART} \]
\[ \text{london} : \text{n} : \text{LOCATION} \rightarrow \]
\[ \text{london} : \text{n} : \text{LOCATION} \]
\[ \text{at} : \text{prep} : \text{S\_TIME} \rightarrow \]
\[ \text{at} : \text{prep} : \text{S\_TIME} \]

\[ \text{exit}(101) : \text{GEN\_FROM\_ENTRY} \]
\[ \text{want.to.leave.london.at} : v : \text{WANT} \]
\[ \text{want} : v : \text{WANT} \]
\[ \text{to} : v : \text{NULL} \rightarrow \]
\[ \text{leave.london.at} : v : \text{DEPART} \rightarrow \]
\[ \text{leave} : v : \text{DEPART} \]
\[ \text{london} : \text{n} : \text{LOCATION} \rightarrow \]
\[ \text{london} : \text{n} : \text{LOCATION} \]
\[ \text{at} : \text{prep} : \text{S\_TIME} \rightarrow \]
\[ \text{at} : \text{prep} : \text{S\_TIME} \]

\[ \text{exit}(101) : \text{GENERATE...} \]

During the generation of \textit{you} (not shown here), the reuse instruction with respect to entry [3] is used.
exit(1) : GEN_FROM_ENTRY
   do.you.want.to.leave.london.at : v : WANT
   do : v : WANT
     you : n : INDIVIDUAL →
     want.to.leave.london.at : v : WANT →
     want : v : WANT
     to : v : NULL →
     leave.london.at : v : DEPART →
     leave : v : DEPART
     london : n : LOCATION →
       london : n : LOCATION
     at : prep : S_TIME →
       at : prep : S_TIME

exit(1) : GENERATE...

done(0) : GENERATE
   what.time.do.you.want.to.leave.london.at : v : S_TIME
   what : v : S_TIME
     time : n : S_TIME →
       time : b : S_TIME
     do.you.want.to.leave.london.at : v : WANT →
       do : v : WANT
     you : n : INDIVIDUAL →
     want.to.leave.london.at : v : WANT →
       want : v : WANT
     to : v : NULL →
     leave.london.at : v : DEPART →
       leave : v : DEPART
     london : n : LOCATION →
       london : n : LOCATION
     at : prep : S_TIME →
       at : prep : S_TIME

5.7.3 Focus marking and partial ordering

During production of the analysis tree, the phon component of the sign at each node has been elaborated to contain the string dominated by that node. At the same time, a node $N$ corresponding to lexical heads may be coindexed to a discourse entity $Id$, by the equation $N :< phon pfocus semid >= Id$. The surface tree derived in the previous section, is thus augmented with semantic indices and reuse instructions:

(5.46) $h : h$ : morform : what
   pfocus : ifocus : dg_ifocus
     semid : s_time2
   a : h : morform : time
   a : h : h : morform : do
     pfocus : reuse : eq
Only relevant surface features are shown. The discourse entities \textit{location2} and \textit{discourserole2} were not needed in the mapping, so are dropped. Focus ordering of the remaining discourse entities now takes place. Those entities marked with the feature \textit{ifocus} are most prominent:

\[
REST \prec \{\text{city2}, \text{s\_time2}\}.
\]

The resulting partial ordering, in canonical form, is

\[
\begin{align*}
\text{depart1} & \prec \text{city2} \\
\text{depart1} & \prec \text{s\_time2} \\
\text{want1} & \prec \text{city2} \\
\text{want1} & \prec \text{s\_time2} \\
\text{caller} & \prec \text{city2} \\
\text{caller} & \prec \text{s\_time2}
\end{align*}
\]

Next, the reuse instructions which are relevant to the analysis tree:

\[
\begin{align*}
\text{city2} & \prec (3, \text{neq}) \\
\text{s\_time2} & \prec (3, \text{typeq}) \\
\text{depart1} & \prec (3, \text{par}) \\
\text{caller} & \prec (3, \text{eq})
\end{align*}
\]

are applied to give the ordering

\[
\text{caller} \prec REST \prec \{\text{city2}, \text{depart1}\}.
\]
This in turn is combined with the previously derived canonical form ordering, the latter taking precedence. The result is:

\[
\begin{align*}
\text{want1} & \prec \text{depart1} \\
\text{s\_time2} & \prec \text{city2} \\
\text{caller} & \prec \text{want1} \\
\text{depart1} & \prec \text{s\_time2}
\end{align*}
\] (5.47)

The rule that applies to pronouns and deictics is redundant, because \text{caller} is already maximally demoted in the ordering. Finally, the rule for informativeness places task-central entities before other discourse entities, which in turn take precedence over semantic entities not explicitly in the discourse model:

\[
\text{want1} \prec \{\text{caller, depart1}\} \prec \{\text{s\_time2, city2}\}
\] (5.48)

There are no instances of the \texttt{ONE\_TO\_MANY} relation (as there could have been had the utterance incorporated the subexpression \ldots Paris Charles de Gaulle \ldots ); combining (5.48) with the previously derived ordering (5.47) results in no change to the ordering. Starting from the lowest ordered identifier, \text{caller}, assignment of integers denoting levels in a total ordering is relatively straightforward:

\[
\begin{align*}
\text{caller} & : 0 \\
\text{want} & : 1 \\
\text{depart} & : 2 \\
\text{s\_time} & : 3 \\
\text{city2} & : 4
\end{align*}
\]

After assignment of annotation markers to the constituents marked with these \textit{semids}, the string sent to the synthesiser is the following:

#5#what#qu#time#n#do#avI#you #:lopI#:want#v#:lo#fw#leave#v6#london#n#at#p#s#sdf#.

The low-level prosody generation algorithm then produces the $F_0$ contour shown in Figure 5.17. The graph shown represents a possible realization of the prominence information. I do not discuss the principles underlying the generation of the low-level contour.
Figure 5.17: F0 contour for generated utterance

5.8 Summary

This chapter has demonstrated how the ideas put forward in Chapter 3 and Chapter 4 are implemented within a working system. Firstly the Sundial dialogue manager was described, and related to the formal model of Chapter 3. Next I presented the implementation of the Belief Module, which maintains the discourse model and is responsible for discourse-specific interpretation. Phases in the generation of output: description and linguistic generation, correspond broadly to Levelt’s (1989) distinction between micro-planning, and formulation. Both exploit previous material, linguistic generation being additionally supported by decisions made during description generation. Production of the focus ordering over discourse entities, and ultimately constituents, was then discussed. This derives in part from discourse information, in particular reuse instructions made during description generation and exploited again during linguistic generation. This partial ordering finds its way into annotations on the linearised string sent to the rule-based synthesizer. Finally, a detailed example of the working of these components was given. Further examples, in the form of verbose output from the program, are given in Appendix D.
Chapter 6

Conclusions and Extensions

6.1 Summary of this thesis

In Chapters 1 and 2, I specified the domain of study: the generation of intonation in an information dialogue, and surveyed the work previously done in this and related areas.

Chapter 3 set out to provide a computational account of a speaker engaged in an information dialogue, sufficiently detailed to enable it to be related to observed intonational phenomena. Firstly, the Flight Enquiries corpus was analysed, to determine what essential aspects of information dialogues needed to be modelled. The most basic structural relation in dialogues was taken to be that between initiatives and responses. In the case of information dialogues, a response may satisfy, nearly satisfy or over-satisfy a task-oriented initiative; or it may satisfy a related (but implicit) initiative. Between initiative and response, there may be a number of insertion sequences, for example to seek clarification, or perform repairs. According to many accounts of communication, speakers communicate successfully if listeners are able to recover their intentions, at least in part. Utterances in the corpus were considered in some detail, in order to determine what surface features, if any, could be exploited by listeners carrying out pragmatic interpretation of this kind. Surface cues such as the use of performative verbs, and syntactic marking of mood, were often available, more in the case of initiatives than responses. Otherwise use was made of contextual cues—this was the case especially when the relevant context
was sufficiently recent. It was also found that dialogue acts do not necessarily map neatly onto utterances and turns. A turn may have a multiple function, such as confirming previous material, and enquiring about something new.

The computational account of an information-providing Agent in dialogue is based on the notion of a number of concurrent, communicating processes. Overt linguistic behaviour is then an extension of such an architecture; messages received and sent to the interlocutor can be treated in a similar way to any other messages between processes. The components of the system were defined, using an algebraic specification language, to fit in with observed dialogue phenomena. The Information Component is responsible for managing transactions with a database (or timetable); it may prompt for new tasks, or seek clarification of underspecified ones. The Interpretation Component is responsible for using contextual resources, together with any surface indications, in order to derive a dialogue act description from the input. Incoming responses are contextualised by matching them with records on a response stack, set up at the time of formulation of the corresponding initiatives. Interpreted input becomes encoded as a message belonging to the characteristic input of the component that is specialised to deal with such input. In the case of input relating directly to specification of the task, this is handled by the Information Component. The State Component handles messages concerning the more ritual aspects of the dialogue, such as openings and closings. The Meta Component is responsible for handling confirmations and repairs. All of these components are capable of taking the initiative; a repair may be initiated, for example, if input from the interlocutor is unclear or incoherent. Of particular interest is the Output Component. This serves those processes which require messages to be sent to the interlocutor. Messages may be stored in an internal buffer, and formulated in batches, when an appropriate opportunity to speak arises. Alternatively, complex messages which require packaging into smaller chunks may be delivered over a number of turns; these may be broken up by interactive subdialogues.

Dialogue acts resulting from the interpretation of input from the Caller are defined to be expressions which are sufficiently specified to act as input to the Information, Meta or State components. The Agent’s dialogue acts are defined reciprocally;
the characteristic input to the output component is therefore equivalent to the characteristic output of the pragmatic interpretation component. These expressions may be augmented by features resulting from decisions taken during production of output, for example those concerning buffering or chunking. The result is a taxonomy of dialogue acts, consisting of featural descriptions which are more or less specified. The assumption is made that textual and prosodic choices made by the speaker aid the listener to recover some or all of the intended features.

The Swedish Airlines corpus was analysed to investigate the possible relation between dialogue acts so defined, and intonational contours. For the speakers analysed, a number of loose generalisations were possible. Initiatives show a greater variation in nuclear contour. Query initiatives are often accompanied by a fall-rise; in the case of my speakers, default queries always have this contour. Contour might for example be used to tell the difference between an open query, and an open confirmation; both can be syntactically wh-questions. Open queries have both falls and fall-rises, and a variety of heads; open confirmations however have fall-rises and high rises, and a tendency to upstep. Other possible generalisations concerned phatic utterances (such as greetings), which tended to use stylised contours, and alternatives questions, which employ list intonation with continuation markers as delimiters. In the case of reformulations or repetitions, when these were compared with their antecedents, increased prominence of accents was found, with the possibility of local contours being modified. Those correlations that were found were however did not amount to a fully determinate account of intonation contours within the model. Thus a complete explanatory theory, for example one in which contours might be used to resolve ambiguities of pragmatic interpretation, was not forthcoming. Instead it would appear that additional factors outside the scope of the model, such as speaker's attitude, may be involved.

Chapter 4 investigated possible mechanisms underlying sentence accent decisions, or prosodic focussing. I proposed mechanisms at three levels. Firstly, at the linguistic level, the focussing decision can be based on comparison with recent linguistic structures. This is of potential benefit to speaker and listener, reducing processing effort. I surmised that accentual patterns may serve to direct the listener to locate
such reused material within the speaker's utterance. Secondly, relative focal prominence may reflect accessibility within the speaker's discourse model. A knowledge representation language was introduced, capable of describing the content of the discourse model, as well as specifying updates to that content. The discourse model itself is represented as a semantic network. Examination of the Swedish Airlines corpus suggested that an ordering based on recency and inferrability might be used in the default case where discourse entities were not intentionally marked. Prosodic focus is defined to be a partial ordering over the discourse entities appearing in the description graph for an utterance, which reflect relative focal prominence. An accessibility history was defined, in a manner analogous to the linguistic history, to be a limited buffer of previous input or output graphs. Focal prominence depending on accessibility of recent material is then derived by comparing the current graph with previous ones. Pronouns, deictic expressions and considerations of relative informativeness also contribute to the prosodic focus. However I argued that their influence is subsidiary to that of historical focus. Contrastive focus was said to occur when the need arose to refer to discrepancies in the discourse model. Examination of the corpora revealed that emphatic prominence was commonly given to the changed material. Conflicting information states can be represented by extending what is common to both differently at different world indices. It is then possible to describe graphs whose purpose is to refer to a discrepancy, and stipulate the component of the focus ordering due to contrastiveness. A similar technique was applied to the case where alternative instantiations of an interlocutor's requirements were described. Here, the speaker could additionally impose rhetorical focus, presenting the information as a functional mapping between its parts. Finally, focal prominence may be said to be derived at the level of the speaker's intentions. Intended components of an utterance, such as information requiring confirmation, may be important and hence require prominence. Discourse entities marked for intentional focus may be associated with levels of emphasis, and particular tonal accents. Taking a different approach to the analysis of pitch movement to that of Chapter 3, I proposed that local pitch movements may serve to distinguish how the information made prominent is to be interpreted. The type of pitch movement accompanying the
prominence could be related for example to whether or not a change of information was authorised.

In Chapter 5 the implementation of the Sundial dialogue manager was presented. Work carried out by others was discussed briefly, the overall architecture being related to the formal model of Chapter 3. Those components of the implementation which relate directly to the derivation of prosodic focus and local contour were then examined. The Belief Module maintains a contextual record of discourse objects and relations, using these in the context-dependent interpretation of input utterances, and deriving output descriptions which exploit context. A conceptual description is obtained from dialogue act specifications, as a graph which covers the intended content, and which is attachable relative to an accessibility history of previous input or output descriptions. The graph-building routines make use where possible of material from the accessibility history, either directly, or by using structural parallels.

Linguistic generation (or formulation) of output starts with a description graph, and derives an analysis tree whose terminals are words or morphemes. The (head-driven) generation algorithm makes use of a declarative lexicon based on Unification Categorial Grammar, ensuring that the analysis tree meets lexical constraints as well as constraints imposed by the input. A linguistic history of previous analysis trees is maintained; portions of former analysis trees can then be re-used where appropriate. Re-cycling of former structures is partially guided by reuse instructions, which record decisions made during the description graph derivation.

Prosodic focus as a relational structure on discourse entities is obtained by a combination of partial orderings. Discourse entities are tagged for intentional focus if singled out in the original dialogue act description. Special cases are where there is a conflict between descriptions, and where rhetorical focus is being applied. Intentional focus takes precedence over orderings based on accessibility. Next is a historically-derived ordering based on successful applications of reuse instructions. Other partial orderings have lower precedence; for example, relative informativeness, which is determined by partitioning discourse entities according to semantic type. The analysis tree contains full featural descriptions of all of its nodes; it is therefore possible to define a functional mapping from discourse entities onto constituents. The
focus ordering on constituents may be supplemented by purely surface ordering decisions. The partial ordering on constituents is mapped onto a total ordering; this is in turn converted to a set of annotation symbols indicating prosodic prominence, to form part of the linearised string sent to the rule-based synthesis component. Finally a detailed example of utterance production was presented.

6.2 Original contributions of this thesis

In this section I explain how this thesis extends existing work so as to make a new contribution to research in the field. I do this by examining the differences and similarities with other approaches.

6.2.1 Dialogue architectures and dialogue acts

The dialogue architecture presented in Chapter 3 is in contrast to that of systems which base interpretation on plan recognition, and production on plan elaboration (Sections 2.2.2, 2.4.2). Whereas these make use of hierarchical plans, in the architecture of AGENT only a shallow representation of intentionality is used. The difference may be illustrated by consideration of Example (2.8), repeated here:

(2.8) patron: when does the Windsor train leave? 
clerk: 3:15, at gate 7.

In the system of Allen and Perrault (1980), the overloaded response: *gate seven* can be explained by the clerk assuming a plan which has the patron needing to know the gate number in order to catch the train. In AGENT this information would have to be returned as part of an overloaded response by the Information component; similarly, in SUNDIAL unsought-for information about the terminal is routinely returned by the Task module. In both there is the implicit default assumption that the Caller would find this information useful. In this respect these accounts might be said to lack explanatory power; certainly the Belief module (Chapter 5) could usefully be extended with a set of defaults about what users are likely to find useful information. That goes beyond the scope of this thesis. Moreover,
examination of the Flight Enquiries corpus demonstrates that human Agents give
certain information, for example about terminal numbers, as a matter of routine,
even when it is clear that many Callers have no use for this information.

AGENT is based on an architecture of communicating processes, or 'homunculus'-
like agents; such an approach was pioneered for natural language dialogue in SUN­
DIAL. AGENT goes beyond this in being specified in a homogeneous algebraic
notation, CSP, for which a considerable body of techniques and results exists. In
addition, processes defined in terms of their characteristic alphabets are a plausible
model of Levelt's (1989) principle of information encapsulation, which requires that
relatively autonomous processes each operate on a characteristic input. I have also
indicated how the specification can be extended to give a partial representation of
the interlocutor. This is in contrast to other dialogue architectures such of that of
Allen, where the internal representation of the interlocutor is quite different from
the representation of self. In this respect it is more like the architectures of Power
(1979) and Houghton and Isard (1986); however, these require that computational
processes for both agents be present.

The representation of dialogue acts presented here is featural. Typical of those
commonly used by computational linguists is that of Cohen (1978), where only a
finite number of dialogue act labels are proposed, each being defined as equivalent
to a number of conditions on the beliefs and intentions of agents. A featural rep­
resentation has the advantage of productivity; for example, closely related dialogue
acts may be distinguished by a single feature, such as repeated. It can also account
for under-specification, leading to ambiguity in pragmatic interpretation, such as
might occur for example when a speaker fails to make clear the difference between
an open query and an open confirmation initiative. I have indicated how it might be
given a semantics in terms of constraints on sequences of events within the dialogue
architecture (Section 3.3.7).

The view of dialogue structure that emerges is similar to that of many ac­
counts which treat the exchange as an important structural primitive (Sinclair and
these, and in contrast with purely Searlian approaches which attempt to characterise
acts in isolation, moves such as responses are treated as relationally dependent on other moves. By extending the basic initiative-response sequence to include pre-initiatives and post-responses, the notation may be applied to longer sequences such as those containing evaluative moves (Moeschler 1989).

The notation is sufficiently flexible to allow features emerging at a number of levels to coexist. It is possible to account for the fact that dialogue acts and turns may not necessarily coincide, using the binary features \textit{dactfinal} and \textit{turnfinal}. A major motivation for requiring this amount of detail has been the attempt to find correlations between combinations of features and intonational contours. Given the variation in intonational practice not only between dialects, but between individuals, a fully systematic treatment may need to wait until sufficiently abstract representations of intonation are proposed. The current study is motivated by the more modest task of finding appropriate intonation for a computer speaker. The featural representation of dialogue acts has been exploited in an attempt to state regularities.

6.2.2 Discourse representation and prosodic focus

The knowledge representation mechanisms presented in Chapters 4 and 5, like many in natural language systems (e.g. Winograd 1972) treat discourse knowledge as part of a nonmonotonically changing database of facts. Unlike those that use a minimal set of propositional axioms, thus requiring closure and consistency to be regularly recomputed (e.g. Jackson 1987) my representation is semantic-network based. Together with the reification of entities representing events and states, this has the advantage that persistence of objects, needed for accounts of prosodic focus and anaphora, is guaranteed. In contrast with Cohen's (1978) representations, which require that beliefs be formulated within a modal logic framework, I follow Garrod and Anderson (1987) in assuming that agents are not particularly cautious in model-building, but have to be prepared to backtrack or negotiate when assumptions go wrong. Thus the Belief module is designed to maintain a common core of accepted beliefs; where agreement is lacking it stores the divergences at different world indices,
pending negotiation.

The assumption of a discourse model, while common to natural language practitioners and psycholinguists, is rarely made explicit by prosodic phonologists in search of semantic and discoursal explanations for prosodic focus. Many accounts (eg, Gussenhoven 1983, Bird 1991) continue to follow Chomsky (1971) and Jackendoff (1972) in characterising prosodic focus in terms of lambda abstraction, using the terminology of focus and presupposition. In other words, prosodic focus simply represents the move from an open proposition, either real or presumed, to an instantiated one. The outcome is a binary feature *focus*, appearing both on semantic representations and surface constituents. In contrast, I adopt a relational representation of focus. This has a number of advantages. Absolute judgements of focus are often hard to make; some constituents for example bearing pre-nuclear accents may nevertheless be judged as lacking in prominence with respect to everything else. The use of a relational rather than binary representation also opens the possibility (not followed here) of more fully exploiting relational representations of prosodic structure (eg, Ladd 1986). It also makes more sense in processing terms: a relational representation will be more sensitive to the many influences which, as I have shown, can affect focus.

In the model proposed here, the need to promote or demote discourse entities (and ultimately surface constituents) in the focus ordering may arise at a number of levels. At the levels of discourse planning and linguistic generation, the mechanisms arise as emergent features of the independently-motivated need to exploit recent context during language production. In deriving prosodic focus, particularly at the discourse level, I have made use of Bolinger's (1986) principle that accent follows interest. *Contra* Gussenhoven (1984), Selkirk (1984) and many prosodic phonologists, there is no direct causal connection between the defocussing of verbs, and the fact that they appear as predicates in conventional logical representations. In *syl.,* both correspond to discourse entities. Moreover, a discourse entity may be opposed to another belonging to a parallel structure, and so be promoted in the focus ordering. The view of context which is required to support this reasoning is layered rather than one-dimensional. By contrast, constructed examples in the literature too often
take 'context' to be just the previous turn or statement.

My treatment of prosodic contrast goes beyond much of the discussion in the literature, which attempted to class cases on the basis of prosodic similarity. Like Couper-Kuhlen (1984) I separate the form and function issues, dealing mainly with the function. My treatment distinguishes two major kinds types of contrast: firstly, where parallel representations are juxtaposed, and secondly where discrepancies in belief states arise. In the latter case I have distinguished between circumstances when either Caller or Agent has proposed the modification, on an authorised or unauthorised basis. I have also provided an explanation for patterns of prominence which signify a kind of theme-rheme organisation; this I suggest may be based on a speaker's decision to present information in terms of some functional relationship.

6.2.3 Prosody in a computational model of language production

The language production architecture described and implemented extends contemporary work into the area of prosodic production. Other accounts, such as that of Dale (1990) have made use of representations of discourse, but for the purpose of handling purely textual phenomena such as anaphora and ellipsis. My system also exploits contemporary computational theories of grammar and the lexicon in a novel way. Sign-based representations such as HPSG (Pollard and Sag 1989) have been of considerable practical use to computational linguists; categorial grammars, it has been suggested (Steedman 1990, Moortgaat 1988) provide plausible accounts of the mismatch between surface structure based on syntactic derivation, and prosodic structure. I have exploited Unification Categorial Grammar differently. Firstly, I have used the mapping between semantic and syntactic constraints to map patterns of prominence over discourse entities onto surface constituents. Secondly, I have demonstrated how the representation can be used not only for intra-sentential relations, but can also be extended to inter-sentential (discoursal) relations, especially those involving echo or parallelism. To this end, I have made use of the bidirectional, declarative nature of the representation, to achieve homogeneity between the datas-
structures built by the parser and those of the generator. The record of linguistic representations is thus available for use by both components.

Computational treatments of prosody have largely been confined to the field of text-to-speech. Researchers there have paid relatively little attention to prosodic focus. Silverman (1987) used a bounded buffer of recent lexical forms as a basis for defocussing words. This behaviour could be modelled by a reduced version of my system, which made use only of equality-based comparisons within the linguistic history. Hirschberg (1990) goes further in the representation of discourse, attempting to build hierarchical structures like those proposed by Grosz and Sidner (1986); prosodic defocussing then plays a similar role to pronominalisation in Grosz and Sidner's work. Defocussing and pronominalisation may be similar in introducing attenuated forms as a result of increased accessibility; they are not however altogether comparable, especially if we accept the relational treatment of the prosodic focus which Bolinger (eg. 1985) has argued for. My work represents a tentative first attempt at a computational model of Bolinger's ideas on focus.

Prosodic generation in a text-to-speech system must make do with a relatively impoverished set of cues; the automatic understanding of unrestricted written language will have to advance considerably before this situation improves. Silverman's and Hirschberg's models both suffer from this drawback. In my model, language is produced, as well as prosody being generated. Working in the area of dialogue (the 'archetypal form of speaking' according to Levelt 1989), and with a simple subworld, I have been able to isolate and account for a number of phenomena, such as prosodic contrast, which would be harder to disentangle were unrestricted language and domains being used. While such simplification leaves a lot of future work still to be done, it is immediately applicable in real systems, where natural language dialogue is being used as a means towards achieving automated information transactions. The Sundial system is an example of this. The system of Houghton, Isard and Pearson (Houghton and Isard 1986, Houghton and Pearson 1988, Isard and Pearson 1988) has also attempted to produce prosody as part of language production within dialogue. Unlike Sundial, which is application-oriented, their system incorporates a computational theory of knowledge and action. However it is not clear how prosodic
decisions relate to the internal representations of the participants. In my model on the other hand, relational decisions at the levels of intention and the discourse model are mapped in a simple fashion onto the focus pattern on the utterance produced. Moreover, the very mechanisms involved in production are in part responsible for those decisions.

6.3 Extensions

6.3.1 Other aspects of intonation

Information dialogues are a convenient domain in which to develop the limited theory of intonation presented here; turns are typically short, so that intonational phrasing is less of an issue than it might be with other genres of dialogue. My theory permits multiple dialogue acts within a turn to be separated by phrase boundaries, and lexically introduced boundaries (such as those associated with conjunctions) to occur act-internally. Such simplifications ignore the role of prosodic structure at a phonological level, in determining phrase boundaries (eg. Giegerich 1985, Ladd 1990). Such analysis is outside the scope of this thesis.

Likewise, a theory of intonation in discourse should account for structure at a macro-prosodic level, that is, above the level of the intonational phrase (eg. Couper-Kuhlen 1983, Couper-Kuhlen 1986). Intonational paragraphs, for example, are often said to begin at a high register setting, and end low in the speaker’s register. Some evidence for topic-initial high key has been found in the SA corpus; instances of high key are not however restricted to topic-initial position, but may occur for example with certain repair initiatives. It is possible to drive a theory of macroprosody off some intentionally structured representation of an extended text, as Hirschberg and Pierrehumbert (1986) have attempted to do. Further examination of data would be required to determine what regularities there are, if any, in the case of information dialogues.

272
6.3.2 The place of prosody within a cognitive theory of language production

In Chapters 4 and 5, I presented an account of prosodic focus in which this was linked to other cognitive mechanisms. It was possible to correlate certain phenomena with levels in the production process. This approach could be extended downwards, tracing the effect, say, of metrical constraints at the phonological level in the final accenting decision. However, the other aspect of prosody treated in this thesis—intonational contour or melody—is less straightforward to situate within a levels account of production. This is partly because examination of context in this case is much less informative. There is insufficient evidence for theories such as that of Brazil, Coulthard and Johns (1980), which would have an intonational structure above the level of the turn emerging over time. My model could however be extended to include a number of constraints on the production of contours. Firstly, we have seen that a number of regularities appear to exist for decisions made at the intentional level, for example, the use of a fall-rise in the case of default queries. At this level might be added the speaker’s attitude; whether or not for example the speaker wishes to appear polite will make a difference in the contour chosen. At the level of description planning, the states of certain discourse entities may result in their association with specific local contours; for example the deep fall-rise may be used when a strong default has been overturned. At the level of lexical access, stereotyped phrases such as I beg your pardon will have equally stereotypical contours associated with them (Bolinger 1986). Another example is the falsetto register which often accompanies certain politeness forms, such as I wonder if you could tell me .... Also at a lexical level, or below it at a phonological level, decisions are made about internal boundary markers, such as continuation rises.

Reformulations represent one case where the influence of context appears to have an effect on intonational contour. The previous contour may be modified, either by the addition of accents of power, or by the choice of a new contour. It might therefore be argued that the previous contour is stored temporarily, in much the way that previous analysis trees are.
6.3.3 Discourse focus

The notion of structural similarity, giving rise at the same time to focal highlighting, and to defocussing, has been exemplified only with semantic forms which are structurally very similar. Inferential similarity, as in the case of the well-known example:

(6.1) John called Henry a republican, then he insulted him.

would require extensions to the mechanism of comparing forms. Similarly, in the corpus

(6.2) SA4:A7: there's a flight this evening at nine.
SA4:A8: arrive Rome eleven thirty

The tendency of speakers is to accent both arrive and Rome, in much the same way that occurs when the previous utterance is the structurally closer depart .... This again would indicate that the highlighting is not purely structural; besides the previous graph in the accessibility history, closely connected material is also considered to be relatively accessible.

In this thesis I have attempted to characterise mechanisms in terms of rules that operate on discrete symbolic representations. An alternative approach would be to model discourse accessibility in terms of patterns of activation over a memory network. The two approaches could then be compared for explanatory power.

6.3.4 Empirical studies

Although this work has been supported by detailed examination of corpora, that evidence alone is insufficient. The Flight Enquiries corpus is too rich and diverse to enable systematic study of the isolated phenomena considered here; the Swedish Airlines corpus lacks spontaneity. Studies presently being carried out within the Sundial project are aimed at consolidating the exploratory conclusions reached here.

House and colleagues (House et al 1992, House and Youd forthcoming) carried out an experimental evaluation in which subjects were asked to judge the appropriateness to context of synthesized utterances presented in the context of mini-dialogues.
The mini-dialogues, together with dialogue act labelling, were derived from the Swedish Airlines corpus. Two conditions were used. In the first, the transcription of the target utterances was enriched to give intonation contours approximating to those found in the Swedish Corpus. The other condition employed default text-to-speech intonation. It was found that in most cases enriched intonation was rated the better. There was strong preference in particular for two dialogue act types: reformulations and unauthorised corrections.

Default and enriched prosody are also being used as conditions in ‘Wizard of Oz’ simulation experiments being carried out at the University of Surrey. In these experiments (Fraser and Gilbert 1991), subjects are encouraged to believe they are having a dialogue with a machine, whereas in fact the computer’s responses are produced by a human agent. When text-to-speech is used to produce the Wizard’s responses, the resulting dialogues may be analysed to find what pragmatic effects emerge out of different prosodic choices. As other aspects of the technology mature, these simulations may be replaced by user trials of a complete system, and the question again asked: which prosodic choices contribute towards successful information transactions, or to the perceived attractiveness of the system.

6.4 Conclusion

In their implementation, the ideas encapsulated in this thesis form only part of a human-computer dialogue system. Once talking machines become available over the telephone, information services are likely to benefit an increasingly wider population of human users. The ultimate success of such systems depends on the steady but slow maturing of speech recognition technology. Further progress in our understanding of how humans use language to negotiate information is also essential (Oviatt and Cohen 1991).

To inspire confidence, such systems will need qualities of understanding and reasoning; they will also need to speak in a way that human interlocutors feel comfortable with. Work is still needed to improve the naturalness and contextual appropriateness of generated output. It is hoped that the ideas presented in this
thesis, relating prosodic choices to internal states during language production, will contribute in a small way towards that long-term goal.
Bibliography


Appendix A

The Swedish Dialogues

The annotations here are abbreviated, usually by omitting the feature label. However, if the feature has value ‘+’, only the feature label is included. If the value is ‘-’, the full label: value pair is shown. Default values are generally not shown. Note that turns, not dialogue acts, are numbered.

Dialogue 1

[SA 1:A1] Swedish airlines  
init; opening

[SA 1:A1'] can I help you  
preinit; task; open

[SA 1:C1] I'd like to reserve a flight to paris on monday evening  
init; update

[SA 1:A2] are you travelling from london  
init; query; default

[SA 1:C2] yes, heathrow  
resp; query; overload

[SA 1:A3] from london heathrow  
init; confirm; value

[SA 1:C3] yes  
resp; confirm

[SA 1:A4] hold on a moment please  
init; hold

[SA 1:C4] thank you  
resp; hold

[SA 1:A5] there are flights at seven, eight fifteen, and ten thirty preinit; alts; update

[SA 1:C5] what time does the eight fifteen arrive  
init; query; open

[SA 1:A6] nine twenty  
resp; query; open

[SA 1:C6] book me on that one please.  
init; update; — topini

[SA 1:A7] what is the name of the passenger  
init; query; open

[SA 1:C7] monroe, em oh en ar oh ee  
resp; query; open

[SA 1:A8] do you want smoking or non smoking  
init; query; alts

[SA 1:C8] I'm sorry  
init; repeat

[SA 1:A9] do you want smoking or non smoking  
init; query; alts; repeated
[SA 1:C9] smoking please

[SA 1:A10] your flight is booked for monday the nineteenth at eight fifteen

[SA 1:A10'] do you want any further transactions

[SA 1:C10] that's all

[SA 1:C10'] thanks very much

[SA 1:A11] you're welcome

[SA 1:A11'] bye bye

Dialogue 2

[SA 2:A1] Swedish airlines

[SA 2:A1'] what information do you require

[SA 2:C1] has SA512 from Amsterdam landed yet

[SA 2:A2] 512 from Amsterdam

[SA 2:A2'] please hold on

[SA 2:C2] thank you

[SA 2:A3] the flight is delayed it's due to arrive at nine fifteen

[SA 2:C3] sorry. what time

[SA 2:A4] the flight is due in at nine fifteen

[SA 2:C4] nine fifteen

[SA 2:A5] that's right

[SA 2:A6] bye bye

[SA 2:C6] bye

Dialogue 3

[SA 3:A1] flight information

[SA 3:A1'] can I help you

[SA 3:A1"] hello

[SA 3:C1] oh hello

[SA 3:C1'] I was ringing to enquire about the flight SA 512 from paris. Is it on schedule

[SA 3:A2] 512 from paris
[SA 3:C2] sorry, 513                         init; correct
[SA 3:A3] 513 from paris              init; confirm; value; mod
[SA 3:C3] that’s right                   resp; confirm
[SA 3:A4] I’ll just check for you (sir/madam)     init; hold
[SA 3:A4'] yes 513 from paris is running according to schedule  resp; query; topic
[SA 3:A4"] do you require any further information     preinit; query; open
[SA 3:C4] that’s all thank you very much             init; preclosing
[SA 3:A5] that’s all right                         resp; preclosing
[SA 3:C5] bye bye then                           init; closing
[SA 3:A6] bye                                        resp; closing

Dialogue 4

[SA 4:A1] swedish airlines flight information  init; opening
[SA 4:A1'] can I help you                      preinit; dgni; task; open; topic
[SA 4:C1] when is the next flight to rome please      init; query; open; topic
[SA 4:A2] are you travelling from heathrow           init; query; default
[SA 4:C2] no stansted                              resp; query; overload
[SA 4:A3] travelling from stansted                 init; confirm; value
[SA 4:C3] sorry what was that                     init; repeat
[SA 4:A4] you’re travelling from stansted           init; confirm; value; repeated
[SA 4:C4] that’s right                             resp; confirm
[SA 4:A5] hold on please,won’t be a moment           init; hold
[SA 4:C5] thank you                                resp; hold
[SA 4:A6] hello                                    init; resume
[SA 4:C6] hello                                    resp; resume
[SA 4:A7] there’s a flight this evening at nine     resp; query; topic; – dfinal
[SA 4:C7] nine oclock                             init; confirm; value
[SA 4:A8] arrive rome eleven thirty               resp; query; open; topic
[SA 4:C8] seven thirty                            init; confirm; value
[SA 4:A9] no eleven thirty                         init; correct
Dialogue 5

[SA 5:A1] swedish airlines

[SA 5:A1'] would you like flight information, or reservations preinit; dgini; task; alts; topic

[SA 5:C1] I'd like to reserve a return flight london stockholm

[SA 5:A2] london to stockholm

[SA 5:A2'] what time do you want to leave

[SA 5:A3] I'll just check for you (madam/sir), please hold on

[SA 5:C3] thanks

[SA 5:A4] sorry to keep you waiting

[SA 5:A5] hello

[SA 5:A6] yes, depart london eight forty five

[SA 5:A7] arrive stockholm ten thirty five

[SA 5:C7] ten thirty five

[SA 5:C7'] thanks very much indeed

[SA 5:A8] you're welcome

[SA 5:A8'] bye bye

[SA 5:C8] bye
Dialogue 6

[SA 6:A1] swedish airlines

[SA 6:A1'] would you like to make a reservation

[SA 6:C1] yes to paris on monday

[SA 6:A2] from london

[SA 6:C2] pardon

[SA 6:A3] are you flying to paris from london

[SA 6:C3] yes monday morning

[SA 6:A4] on monday in the morning

[SA 6:C4] yes

[SA 6:A5] just hold on a moment please

[SA 6:C5] thanks

[SA 6:A6] hello

[SA 6:A6'] hello there

[SA 6:C6] hello

[SA 6:A7] there are flights at eight forty nine fifty and ten twenty is any of these convenient for you

[SA 6:C7] can you book me on the eight forty please

[SA 6:A8] eight forty to paris

[SA 6:A8'] hold on

[SA 6:A8"] do you want to travel business or economy class

[SA 6:C8] say that again please

[SA 6:A9] do you want to travel business or economy class

[SA 6:C9] business class please

[SA 6:A10] business class

[SA 6:A10'] okay (madam/sir) you are booked on flight SA 197

[SA 6:C10] could you repeat the flight number

[SA 6:A11] SA one nine seven

[SA 6:C11] thanks a lot

[SA 6:A12] thank you

[SA 6:A12'] bye

[SA 6:C12] bye bye

292
Dialogue 7

[SA 7:A1] flight information good day init; opening
[SA 7:C1] yes hello good morning resp; opening
[SA 7:C1'] when is the next flight to rome please init; query; open; topic
[SA 7:A2] from heathrow init; query; default
[SA 7:C2] sorry init; repeat
[SA 7:A3] are you travelling to rome from heathrow init; query; default; repeated
[SA 7:C3] no stansted resp; query
[SA 7:A4] travelling from stansted init; confirm; value
[SA 7:C4] yes resp; confirm
[SA 7:A5] would you hold on please init; hold
[SA 7:C5] okay thank you resp; hold
[SA 7:A6] hello init; resume
[SA 7:A6'] hello init; resume; repeated
[SA 7:C6] hello resp; resume
[SA 7:A7] there's a flight this evening at nine resp; query; — dfinal; open; topic
[SA 7:C7] nine oclock init; confirm; value
[SA 7:A8] arrive rome eleven thirty resp; query; open; topic
[SA 7:C8] eleven thirty init; confirm; value
[SA 7:C8'] thank you very much init; preclosing
[SA 7:A9] thank you resp; preclosing
[SA 7:A9'] bye init; closing
[SA 7:C9] bye resp; closing

Dialogue 8

[SA 8:A1] swedish airlines init; opening
[SA 8:A1'] good day init; opening
[SA 8:A1"] how can I help you preinit; query; open; topic
[SA 8:C1] is there a flight on sunday morning, london to stockholm init; query; value; topic
[SA 8:A2] london to stockholm init; confirm; value
[SA 8:C2] sorry I didn’t get that

[SA 8:A3] you want a flight from london to stockholm on sunday morning

[SA 8:C3] that’s right

[SA 8:C3'] is there a flight around mid morning

[SA 8:A4] I’ll just find out for you (madam/sir) please hold the line

[SA 8:C4] thanks

[SA 8:A5] won’t be a moment

[SA 8:C5] that’s all right

[SA 8:A6] hello

[SA 8:C6] hello

[SA 8:A7] well, there’s a flight at eight fifteen

[SA 8:C7] is that the only one in the morning

[SA 8:A8] yes

[SA 8:C8] okay then

[SA 8:A9] do you want to reserve a seat

[SA 8:C9] no thanks

[SA 8:C9'] thanks very much then

[SA 8:A10] thank you

[SA 8:A10’] goodbye

[SA 8:C10] bye

**Dialogue 9**

[SA 9:A1] good day

[SA 9:A1'] swedish airlines flight information

[SA 9:A1”] hello

[SA 9:C1] hello

[SA 9:C1’] do you know the expected arrival time of flight SA153


[SA 9:A2’] where is that from

294
[SA 9:C2] sorry

[SA 9:A3] where is flight SA153 coming from

[SA 9:C3] geneva

[SA 9:A4] from geneva

[SA 9:A4'] I'll just check that for you (madam/sir). hold on please

[SA 9:A4''] there isn't a flight one five three from geneva

[SA 9:A4''] there's a flight one nine three

[SA 9:C4] it must be that then

[SA 9:A5] the flight is on schedule. It's due to arrive at Heathrow

[SA 9:C5] ten twenty five

[SA 9:A6] that's right

[SA 9:C6] thanks very much

[SA 9:A7] thank you

[SA 9:C7] bye bye

[SA 9:A8] bye

295
Appendix B

Contour transcriptions from the Swedish Dialogues

Inits

Updates:  [seqlab: init, type: update]

[SA 5:C1] I'd 1like to reserve a 2return flight 3 4london stockholm 5 on 6january twenty 7third

JM 1H 2HLH 3] 4HLH 5] 6H 7] 1HL
MG 0[l 1H 2HLH 3] 4H 6H 7] 1HL

[SA 1:C1] I'd 1like to reserve a 2flight to 3paris on 4monday 5sevening

JM 1H 2] 3HLH 4H 5] 1HL
JQ 0[l 1H 3] 1HL] 4H 5] 1HL
MC 0[l 1H 3] 1HL] 4H 5] 1HL
MG 0[l 1H 3] 1HL] 4H 5] 1HL

[SA 6:C1] yes to 2paris on 3monday

JM 1HL] 2HL] 3HL
MG 1HL] 2H 3HL

[SA 6:C7] can you 1book me on the eight 2forty 3please

MG 0[l 1H 2] 1HL 3LH

[SA 1:C6] book me on 2that one 3please

JM 1H 2] 1HL 3LH
MG 0[l 1H 2] 1HL 3LH

Downstepped falls are common, especially if the politeness marker LH please is analysed as a distinct intonational phrase.

Open queries:  [seqlab: init, type: query, qtype: open]

[SA 1:C5] what time does the eight 2fifteen arrive

JM 1H 2] 1HL
JQ 1H 2] 1HLH
MG 1H 2] 1HL
what is the name of the passenger

JM, MG 1 H 2 HLH
JQ 1 H 2 H 3 HLH

what time do you want to leave

JM 1 H 2 H 3 HL
JQ 1 H 3 HLH
MC 1 H 3 HL

when is the next flight to Rome please

JM 0[1' H 3 HL 4 LH 1]
MC 1 H 3 LH
MG 1 H 2' H 3' HL 4 LH

do you know the expected arrival time of flight one five three

JM 0[1' 2 HLH 3] 4 H 5 H 6 HLH
JQ 1' 2 HLH] 5 H 6 HLH
MC 1 H 2 HLH] 4 H 5 H 6 HLH
MG 0[1' (H) 2' HLH 3] 5 H 6 HLH

where is that from

JM 1 H 3' HL 1]
JQ 1 H 3 HLH
MG 0[1' 1 H 2 HLH

where is flight SA one five three coming from

JM, JQ 1 H 2' HL
MG 1 H 2 HLH

Fall-rises outnumber falls, especially if the instances of SA7:C1: are re-analysed as fall-rises. [9:A2] is the repeat of [9:A1]. JM and JQ change contour between repeats; whereas MG simply begins the repeat on a lower key.

Alternatives queries: [seqlab : init, type : query, qtype : alts]

[SA 1:A8] do you want smoking or non-smoking

JM 0[1' 1 H 2' HL 1]
JQ 0[1' 1 H 2' HL
MG 1 H 2' HL

[SA 1:A9] do you want smoking or non-smoking

JM 0[1' 1 HLH] 2 HL
JQ 1 HLH] 2' HL
MC 1 HLH] 2 HL
MG 1' LH] 1' 2' HL

[SA 6:A8”] do you want to travel business or economy class

JM 2 HLH] 3' HL 1]
JQ 2 H] 3' HL
MG 1 H 2 H] 1' 3' HL
All speakers use some kind of continuation tone on the turn-nonfinal components, and finish with a fall. The repeats all show some change in contour; the general effect seems to be to emphasize the phrasing the second time round.

Default queries: [seqlab: init, type: query, qtype: default]
[SA 8:C7] is that the only one in the morning

Default queries mostly finish with HLH nuclei. Two interesting exceptions are [6:A3:JM,JQ] and [7:A3:JM]. In both these cases an initiative is being repeated.

Value Confirmations: [seqlab: init, type: confirm, qtype: value]

[SA 1:A3] from London Heathrow
JM 0[1 1H 2H 3HL
JQ 0[1 1H 2H 3HL
MC 0[1 1H 3HL
MG 1H 2HL

[SA 2:A2] five one two from Amsterdam
JM 1H (2H) 4HLH
JQ 1H 3HL
MC 1H 2H 3HL
MG 1H 2H 3HL

[SA 3:A2] five one two from Paris
JM, MG 1H 3HL
MC 1H 2H 3HL [authd: -]
JQ 1H 2HL

[SA 3:A3] five one three from Paris
JM 2H
JQ 1H 2HL
MC 1H 2H
MG (1H) 2H

JQ is exceptional in not emphasizing the changed element. This must be seen however in the light of his reading of [3:A2], where it seems that he has looked ahead in the script to the upcoming correction, and decided to give an incredulous reading (cf. discussion of Example 4.27 in Section 4.3.4.1, page 167. [3:A3] then becomes a confirmation that he was right in suspecting a mistake in the first place.

JM 0[1 1H 2HL (4,7)
JQ 0[1 1H 2HL 1 (4)
MC 1H 2HL (7)
MG 1H 2HL (4,7)

[SA 4:A4] you're travelling from Stansted
JM, MG 1H 2HL
JQ 0[1 1H 2HL 1]
[SA 6:A4] on Monday in the morning

JM 0[^1] 1H 2^1HL
JQ 0[^1] 1HLH 2^1HLH
MG 1H 2^1HL

[SA 5:A2,8:A2] London to Stockholm

JM 1H 2^1HL(5); [^1] 1H 2^1HL(8)
JQ 1H 2HL(5); 1H 2HLH(8)
MC 1H 2HL(5)
MG 1H 2^1HL(5,8)

[5:A2] clearly differs from [8:A2] in being turn-nonfinal, and hence [strong: -]. This may account for JQ’s use of HLH in [8] but not [5], and JM’s starting on a higher register setting in [8].

[SA 6:A8] Eight forty to Paris

JM 1H 3^1HL[^1]
JQ 1H 2HLH 3HL
MC 1H 3^1HL
MG 1H 2^1HL 3^1HL

[SA 8:A3] you want a flight from London to Stockholm on Sunday morning

[repeated: +]

JM 1H 2^1H 3H 4^1HLH] 5H 6^1HL
JQ 3H 4^1HLH 5H 6^1HLH
MG [^1] 1H 3^1H 6HLH

[SA 9:A2] Less say one five three

JQ 1H 2^1H 3^1HLH
MG 1H 2^1H 3^1HL[^1]


JM, JQ, MG 0[^1] 1HL

Open Confirmations: [seqlab: init, type: confirm, qtype: open]

[SA 2:C3] sorry. What time

JM, JQ 1H 2^1HLH
MG 1HH[^1] 2HH (HH)

[SA 6:C10]: I could you repeat the flight number

JM [^1] 1^H 2^1HLH
JQ 2H 3HLH
MC 1H 3HLH
MG [^1] (2H) 3^1HLH

Corrections: [seqlab: init, type: correct]

[SA 3:C2] sorry five one three

JM 1HLH] 2H 3HLH
MC 1HLH] 2H 3HL
MG 1H 2HLH] 2H 3^1HL

300
[SA 4:A9] 1no el 2even thirty
  JM  1LH 2'HL
  MC  1HL] 2HL

[SA 9:A4"] there 1isn't a flight one five 2three from 3geneva [authorised : -]
  JM  1'HLH 2'HLH
  JQ  1'HL 2HLH
  MC  1'HL 3HL
  MG  1'HL

[SA 9:A4"] there's a 1flight 2one 3nine three [authorised : -]
  JM  1H 2H 3'HL
  JQ,MG 2H 3'HLH
  MC  1H 3'HLH

Repeats:  [seqlab : init, type : repeat]

[SA 1:C8] I'm 1sorry
  MG  0[′ 1'HH ′] [type : hold]

[SA 4:C3] 1sorry 2what was 3that
  MG  1'HH ′] 2H 3HH ′]

Holds and phatics:  [seqlab : init, type : phatic]

[SA 1:A4] 1hold 20n 3a moment 4please  [type : hold]
  JM  0[′ 2̅H 3′̅H 4LH]
  JQ  1H 2′H 4LH
  MC  0[′ 2̅H 4′H]

These are all variants of the calling contour LHM, with optional LH on please.

[SA 2:A2'] 1please 20n 3a moment 4please  [type : hold]
  JM  1H 2'H 3LH
  JQ  1̅H 2′H 3LH
  MC  1H 3LH

Pre-initiatives:  [seqlab : preinit]

[SA 1:A1',3:A1',4:A1'] can I 1help you

  JM  L̅H ′̅H ′] (1); 0[′ 2HLH (3;4)
  JQ, MC, MG  L̅H ′̅H ′] (2;3)
  JQ  2HLH (3)

[SA 2:A1'] 1what infor 2mation do you require
  JM  1̅H 2′HLH ′]
  JQ  1H 2′HL
  MC  0[′ 2HLH
[SA 8:A1'] how can I help you

JM 1H 2HL
JQ 1H 2'H 3'HH
MC 1H 2'HHL
MG 1'H 2'H 3'H

[SA 1:A10'] do you want any further transactions

JM 1'HHLH
JQ 1H 2'HHLH
MC 1H 3'HHLH
MG 1H 2'HHLH

[SA 3:A4'] do you require any further information

JM,MC 0[l] 2'HHLH
JQ 0[l] 1'H 2'HHLH
MG 1'H 2'HHLH

[SA 5:A1'] would you like flight information or reservations

JM 0[l] 2'H 4] 5'H 6'HHL
JQ 2H 3'HHLH 5'H 6'HHL
MG (1H) 2'H 4] 6'HHL

[SA 6:A1'] would you like to make a reservation

JM 1'H 2HLH
MG 0[l] 1'H 2'H 1'H

Responses

topic major responses:  [seqlab : resp, topic : +]

[SA 2:A3] the flight is delayed it's due to arrive at nine fifteen

JM 1H 2'HHL] 3'H 4'H 5'H 6'HHL
JQ 1H 2'HHL] 3'H 4'H 5'H 6'HHL

Both speakers attempt to maximise information transfer by placing accents on every content word. JQ does this in a more emphatic way.

[SA 2:A4] the flight is due in at nine fifteen

JM 1HL 3'H 4'HHL
JQ,MC 1H 2'HHLH] 3'H 4'HHL
MG 1'H 2HL] (3'H) 4'HHL

[SA 3:A4'] yes 2513 from Paris is running according to schedule

JM 1'H] 2H 3HLH 4'H 5'H 6'HHL
JQ 1HLH] 2H 3'H 4'H 5'H 6'HHL
MC 1HL] 2H 3HLH 6'HHL
MG 1HL] 2H 3HLH 4'H 5'H 6'HHL

[SA 4:A7,7:A7] there's a flight this evening at nine  

[-dfinal]
Only MC uses a deferential fall-rise. The other speakers may not however have noticed that a relaxation was involved.

simple Yes/no responses: [seqlab : resp, qtype : default]

[SA 1:C3,8:A8] yes

JM, MG 1HL
JQ, MC 0[1 HL]

[SA 2:A5] that's 2right

JM 1H 2H
JQ, MC, MG 1HL

overloaded yes/no responses: [seqlab : resp, qtype : default, overloaded : +]

[SA 1:C2] yes, 2heath 3row

303
Response to open confirmation:  [seqlab: resp, type: confirm, qtype: open]

All speakers mark the repetition with a pitch accent on every word, and a phrase boundary between the letters and the digits.
## Focus in the Swedish Dialogues

<table>
<thead>
<tr>
<th>Word</th>
<th>Accent Score/4</th>
<th>Distance</th>
<th>Verbal</th>
<th>Intentional</th>
</tr>
</thead>
<tbody>
<tr>
<td>[SA 1:C1] reserve 0</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[SA 1:C1] flight 1.5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[SA 1:A2] travelling 1.5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[SA 1:A3] london 4</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[SA 1:A3] heathrow 4</td>
<td>1</td>
<td>-</td>
<td>TP</td>
<td></td>
</tr>
<tr>
<td>[SA 1:A5] flights 1.5</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[SA 1:C5] 8:15 4</td>
<td>1</td>
<td>-</td>
<td>TP</td>
<td></td>
</tr>
<tr>
<td>[SA 1:C5] arrive 0</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[SA 1:C6] book 4</td>
<td>10</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[SA 1:C6] that 4</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[SA 1:A10] flight 4</td>
<td>7</td>
<td>-</td>
<td>RQ</td>
<td></td>
</tr>
<tr>
<td>[SA 1:A10] booked 4</td>
<td>7</td>
<td>+</td>
<td>RQ</td>
<td></td>
</tr>
<tr>
<td>[SA 1:A10] monday 4</td>
<td>17</td>
<td>-</td>
<td>RQ</td>
<td></td>
</tr>
<tr>
<td>[SA 1:A10] 8:15 4</td>
<td>9</td>
<td>-</td>
<td>RQ</td>
<td></td>
</tr>
<tr>
<td>[SA 1:A10] transactions 1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[SA 2:C1] landed 1</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>[SA 2:A2] 512 4</td>
<td>1</td>
<td>-</td>
<td>TP</td>
<td></td>
</tr>
<tr>
<td>[SA 2:A2] amsterdam 4</td>
<td>1</td>
<td>-</td>
<td>TP</td>
<td></td>
</tr>
<tr>
<td>[SA 2:A3] flight 4</td>
<td>2</td>
<td>-</td>
<td>RQ</td>
<td></td>
</tr>
<tr>
<td>[SA 2:A3] arrive 1</td>
<td>3</td>
<td>+</td>
<td>RQ</td>
<td></td>
</tr>
<tr>
<td>[SA 2:A4] the flight 4</td>
<td>2</td>
<td>-</td>
<td>RF</td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>Accent Score/4</td>
<td>Distance</td>
<td>Verbal</td>
<td>Intentional</td>
</tr>
<tr>
<td>------</td>
<td>----------------</td>
<td>----------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>paris</td>
<td>2.5</td>
<td>1</td>
<td>-</td>
<td>TP</td>
</tr>
<tr>
<td>513</td>
<td>4</td>
<td>3</td>
<td>-</td>
<td>RQ</td>
</tr>
<tr>
<td>paris</td>
<td>4</td>
<td>3</td>
<td>-</td>
<td>RQ</td>
</tr>
<tr>
<td>running</td>
<td>4</td>
<td>-</td>
<td>+</td>
<td>RQ</td>
</tr>
<tr>
<td>schedule</td>
<td>4</td>
<td>5</td>
<td>-</td>
<td>RQ</td>
</tr>
<tr>
<td>information</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>travelling</td>
<td>1</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>travelling</td>
<td>3.5</td>
<td>2</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>travelling</td>
<td>4</td>
<td>2</td>
<td>+</td>
<td>RF</td>
</tr>
<tr>
<td>flight</td>
<td>4</td>
<td>11</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>arrive</td>
<td>4</td>
<td>-</td>
<td>+</td>
<td>RQ</td>
</tr>
<tr>
<td>rome</td>
<td>3.5</td>
<td>13</td>
<td>-</td>
<td>RQ</td>
</tr>
<tr>
<td>thirty</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>RQ</td>
</tr>
<tr>
<td>reserve</td>
<td>0</td>
<td>1</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>flight</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>london</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>TP</td>
</tr>
<tr>
<td>stockholm</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>TP</td>
</tr>
<tr>
<td>leave</td>
<td>4</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>flight</td>
<td>1</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>depart</td>
<td>4</td>
<td>-</td>
<td>+</td>
<td>RQ</td>
</tr>
<tr>
<td>london</td>
<td>4</td>
<td>9</td>
<td>-</td>
<td>RQ</td>
</tr>
<tr>
<td>arrive</td>
<td>2</td>
<td>-</td>
<td>+</td>
<td>RQ</td>
</tr>
<tr>
<td>stockholm</td>
<td>4</td>
<td>11</td>
<td>-</td>
<td>RQ</td>
</tr>
<tr>
<td>flying</td>
<td>1.5</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>paris</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>RF</td>
</tr>
<tr>
<td>monday</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>TP</td>
</tr>
<tr>
<td>morning</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>TP</td>
</tr>
<tr>
<td>flights</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>these</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>book</td>
<td>2</td>
<td>13</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>8:40</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>TP</td>
</tr>
<tr>
<td>paris</td>
<td>4</td>
<td>9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>travel</td>
<td>0</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>travel</td>
<td>0</td>
<td>2</td>
<td>+</td>
<td>RF</td>
</tr>
<tr>
<td>flight number</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>travelling</td>
<td>(2)</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>rome</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>travelling</td>
<td>4</td>
<td>2</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>stansted</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>TP</td>
</tr>
<tr>
<td>flight</td>
<td>4</td>
<td>10</td>
<td>-</td>
<td>RQ</td>
</tr>
<tr>
<td>arrive</td>
<td>4</td>
<td>-</td>
<td>+</td>
<td>RQ</td>
</tr>
<tr>
<td>rome</td>
<td>2</td>
<td>10</td>
<td>-</td>
<td>RQ</td>
</tr>
<tr>
<td>Word</td>
<td>Accent Score/4</td>
<td>Distance</td>
<td>Verbal</td>
<td>Intentional</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
<td>----------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>[SA 8:C1]</td>
<td>flight</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:A2]</td>
<td>london</td>
<td>4</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:A2]</td>
<td>stockholm</td>
<td>4</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:A3]</td>
<td>flight</td>
<td>1</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:A3]</td>
<td>london</td>
<td>4</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:A3]</td>
<td>stockholm</td>
<td>4</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:A3]</td>
<td>sunday</td>
<td>1</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:A3]</td>
<td>morning</td>
<td>4</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:C3]</td>
<td>flight</td>
<td>0</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:C3]</td>
<td>morning</td>
<td>3</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:A7]</td>
<td>flight</td>
<td>2</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:C7]</td>
<td>that</td>
<td>1.5</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:C7]</td>
<td>morning</td>
<td>0</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>[SA 8:A9]</td>
<td>reserve</td>
<td>1.5</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>[SA 8:A9]</td>
<td>seat</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[SA 9:A2]</td>
<td>that</td>
<td>2</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>[SA 9:A3]</td>
<td>SA153</td>
<td>2.5</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>[SA 9:A4]</td>
<td>geneva</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>[SA 9:A4]</td>
<td>flight</td>
<td>0</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>[SA 9:A4]</td>
<td>153</td>
<td>2</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>[SA 9:A4]</td>
<td>geneva</td>
<td>2</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>[SA 9:A4]</td>
<td>flight</td>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>[SA 9:A4']</td>
<td>one</td>
<td>3</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>[SA 9:A4']</td>
<td>three</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>[SA 9:C4]</td>
<td>that</td>
<td>4</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>[SA 9:A5]</td>
<td>flight</td>
<td>4</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>[SA 9:A5]</td>
<td>arrive</td>
<td>2</td>
<td>8</td>
<td>-</td>
</tr>
</tbody>
</table>

307
Appendix D

Example output: description and generation

This appendix contains verbose printout from the running of the description generator/microplanner (Section 5.3) and the linguistic generator (Section 5.4). The lines of output from the latter component are prefixed ‘GEN’. Working together, these components illustrate both the reuse mechanisms, and accent assignment. The first dialogue, nfenq111, is based on the one presented in Section 4.3.3.2. Generation of the utterance A3 from this dialogue was described in detail in Section 5.7. There follow four dialogues which are variants on the first. Only the production of those utterances which are different, and interesting, is shown. Dialogue nfenq511 (page 323) illustrates constraint relaxation by the system, leading to an unauthorised modification. Dialogue nfenq311 (page 331) illustrates the response to an authorised modification by the Caller. Dialogues nfenq611 (page 337) and nfenq811 (page 339) illustrate authorised modifications by the system. The latter dialogue also illustrates deaccenting of pronouns, and the use of ‘referring focus’ for rhetorical marking of theme-rheme. Finally, sa4 (page 343), based on Swedish Airlines dialogue 4, illustrates the response to overturned defaults. The moves correspond to [SA 4:C1–4:A4].

It should be noted that feature attribute structures are unordered, and are printed out as they appear in computer memory. SIL expressions as used by the description generator tend to be large. For this reason, subsequent mentions of these are shortened, and indexed where there may be ambiguity.

<<<<<I FEIQ111
<< Illustrates: baseline dialogue presented in Chapter 4
<<
<< C: i want to travel from London to Paris: init; task; topic
<< A: what time do you want to arrive in Paris: init; query; open

msg(bm,mp,request_description_object_in_ctx(mp5,know(shared,avs(dbflight1,[goalti me:A,goalcity:paris])),B))).

The MP sends the BM a request for a description, in terms of task-oriented parameters. Any additional material required (in this case the query material) has been added to the BM in advance. Using a lookup table which traces inferential
paths between task and surface oriented information, the ids which are required to appear in the description graph (the 'core ids') are found.

The procedure describe_ids_ah now looks in the accessibility history—which at this stage contains only one entry—for description subgraphs that span these ids (and no other material). The subgraph with root location1 is found. The subgraph with root s_time1 has to be built.

*****BEGINNING DESCRIPTION CYCLE******

*** IDs to generate: [minutes1,s_time1,hour1,city1,location1]
*** AH Buffer:
1:
id : go1
type : go
goal : id : location1
type : location
    thecity : id : city1
type : city
        value : paris
source : id : location2
type : location
    thecity : id : city2
type : city
        value : london
theme : id : caller
type : individual
discourse : id : discourse1
type : discourse
    value : speaker

*** Found Subgraphs: [[id:location1,type:location,...] ]
*** Built Subgraphs: [
id : s_time1
type : s_time
hour : id : hour1
type : hour
    value : ????
minutes : id : minutes1
type : minutes
    value : ????
]

The procedure find_spanning_ah now attempts to build a spanning graph which covers the subgraphs found.

The accessibility history entry (root go1) is again considered, since it contains the subgraph with root location1.

*****Building Graph ...
***Subs [[id:location1,type:location,...] ] spanned by new graph
destination : id : single_journey1
type : single_journey
passenger : id : passenger1
type : passenger
    individual : id : caller
type : individual
discourse : id : discourse1
However, this candidate cannot be taken further, because the lexical-tag constraints for type:go do not allow for role fillers of type s.time.

Since there is no other candidate in the accessibility history, the default chaining procedure is used, to find a lexically-admissible root which covers the subgraphs. Other lexically required material (in this case the subgraph dominated by caller1) is added.

**Built new graph:**

```plaintext
thejourney : id : single_journey1
type : single_journey
thepassenger : id : passenger1
type : passenger
theindividual : id : caller
type : individual
thediscourserole : id : discourserole
```

However, this candidate cannot be taken further, because the lexical-tag constraints for type:go do not allow for role fillers of type s.time.

Since there is no other candidate in the accessibility history, the default chaining procedure is used, to find a lexically-admissible root which covers the subgraphs. Other lexically required material (in this case the subgraph dominated by caller1) is added.

**Built new graph:**

```plaintext
thejourney : id : single_journey1
type : single_journey
thepassenger : id : passenger1
type : passenger
theindividual : id : caller
type : individual
thediscourserole : id : discourserole
```

However, this candidate cannot be taken further, because the lexical-tag constraints for type:go do not allow for role fillers of type s.time.

Since there is no other candidate in the accessibility history, the default chaining procedure is used, to find a lexically-admissible root which covers the subgraphs. Other lexically required material (in this case the subgraph dominated by caller1) is added.

**Built new graph:**

```plaintext
thejourney : id : single_journey1
type : single_journey
thepassenger : id : passenger1
type : passenger
theindividual : id : caller
type : individual
thediscourserole : id : discourserole
```

However, this candidate cannot be taken further, because the lexical-tag constraints for type:go do not allow for role fillers of type s.time.

Since there is no other candidate in the accessibility history, the default chaining procedure is used, to find a lexically-admissible root which covers the subgraphs. Other lexically required material (in this case the subgraph dominated by caller1) is added.

**Built new graph:**

```plaintext
thejourney : id : single_journey1
type : single_journey
thepassenger : id : passenger1
type : passenger
theindividual : id : caller
type : individual
thediscourserole : id : discourserole
```

However, this candidate cannot be taken further, because the lexical-tag constraints for type:go do not allow for role fillers of type s.time.

Since there is no other candidate in the accessibility history, the default chaining procedure is used, to find a lexically-admissible root which covers the subgraphs. Other lexically required material (in this case the subgraph dominated by caller1) is added.
Finally the substructure corresponding to the unknown elements is deleted, and a lambda-abstraction is formed.

Extracted s_time1 to make desc:

```
thejourney : id : single_journey1
type : single_journey
thepassenger : id : passenger1
type : passenger
theindividual : id : caller
type : individual
thediscourse_role : id : discourse_role

type : discourse_role
value : hearer
```

```
theplace : id : location1
type : location
thecity : id : city1
type : city
value : paris

thetime : id : time_point1
type : time_point
thetime : id : s_time1
type : s_time
thehour : id : hour1
type : hour
value : ????
theminutes : id : minutes1
type : minutes
value : ????
```

to cover subs [[id:location1,type:location,...],[id:s_time1,type:s_time,...]]

**INPUT SEMANTICS**

```
GEN> id : s_time1
GEN> type : s_time
GEN> modus : def : wh
GEN> thedesc : id : want1
GEN> type : want
GEN> theagent : id : caller
GEN> type : individual
GEN> modus : ref : pro
GEN> thetheme : id : arrive1
GEN> type : arrive
GEN> theplace : id : location1
GEN> type : location
```
Both location (Paris) and caller are reused. However, only the latter gets deaccented, because the other is marked for intentional focus.
The description corresponding to the echo is built by locating the subgraph (root \( s_{\text{time1}} \)) in the accessibility history. Since there is no additional material required for the description, and \( s_{\text{time1}} \) is currently unique, this subgraph is returned as the description graph.

*****BEGINNING DESCRIPTION CYCLE*******

*** IDs to generate: [minutes1, s_time1, hour1]  
*** AH Buffer: 
1: [id: s_time1, type: s_time, ...]  
2: [id: s_time1, type: s_time, thedesc: ...]  
3: [id: go1, type: go, ...] 

*** Found Subgraphs: [[id: s_time1, type: s_time, ...]]  
****Building Graph ...  
****(sub)Graph [id: s_time1, type: s_time, ...] from AH entry 1 spans subs [[id: s_time1, type: s_time, ...]] 

In the knowledge bases currently used, there is a mismatch in granularity between representations in the BM (where hours and minutes are represented as integers) and in the generator (where hours and minutes are decomposed into digits). The input semantics below is a result of converting to the latter representation. The terms \( gi(\text{Type}, \text{No}) \) are newly created identifier symbols.

GEN> *****INPUT SEMANTICS******
GEN> id : s_time1 
GEN> type : s_time 
GEN> thehour : id : hour1 
GEN> type : hour 
GEN> t : id : gi(digit,0) 
GEN> type : digit 
GEN> value : 0 

313
As with the earlier wh-question, the subgraph corresponding to the location is found, while the subgraph for the unknown time has to be built. Note position 2) in the accessibility history simply contains a back-pointer to position 1).

*****BEGINNING DESCRIPTION CYCLE*****

*** IDs to generate: [minutes2.s_time2,hour2,city2.location2]

*** AH Buffer:  
1: [id:s_time1,type:s_time,...]  
2: backref(1,)  
3: [id:s_time1,type:s_time,thedesc:...]  
4: [id:go1,type:go,...]

*** Found Subgraphs: [[id:location2,type:location,...]]  
*** Built Subgraphs: [  
id : s_time2  
type : s_time  
thehour : id : hour2  
type : hour  
value : ?? ??]
The procedure find_spanning_ah searches for a spanning graph. This time entry 3) is used, since its structure corresponds to the structure required. As before, a lambda-abstract is then built.

```plaintext

Thejourney : id : single_journey
  type : single_journey
  thepassenger : id : passenger
    type : passenger
    theindividual : id : caller
      type : individual
      thediscourserole : id : discourserole
  2
le

id : depart
  type : depart
theplace : id : location
  type : location
  thecity : id : city
    type : city
    value : london

parallel to already built [id:arrivel, type: arrive, ...]

Extracted s_time2 to make desc:

thejourney : id : single_journey
  type : single_journey
  thepassenger : id : passenger
    type : passenger
    theindividual : id : caller
      type : individual
      thediscourserole : id : discourserole
  2
le

id : depart
  type : depart
theplace : id : location
  type : location
  thecity : id : city
    type : city
    value : london
```

315
type : location
thecity : id : city2
    type : city
    value : london

GEN> *******INPUT SEMANTICS*********

GEN> id : s_time2
GEN> type : s_time
GEN> modus : def : wh
GEN> thedesc : id : want1
GEN>    type : want
GEN> theagent : id : caller
GEN>    type : individual
GEN> thediscourserole : id : discourserole2
GEN>        type : discourserole
GEN>        value : hearer
GEN> thetheme : id : depart1
GEN>    type : depart
GEN> thetime : id : time_point2
GEN>    type : time_point
GEN> theplace : id : location2
GEN>        type : location
GEN> thecity : id : city2
GEN>        type : city
GEN>        value : london
GEN> thetheme : id : caller
GEN>    type : individual
GEN> thediscourserole : id : discourserole2
GEN>          type : discourserole
GEN>          value : hearer

GEN> *******ALIST*******
GEN> s_time2:[ifocus:dg_ifocus.owner:inferred(system),reuse:[ri(3,typeq)]].
GEN> depart1:[owner:inferred(caller),reuse:[ri(3,par),new]].
GEN> time_point2:[owner:inferred(system),reuse:[new]].
GEN> location2:[ifocus:dg_ifocus.owner:caller,reuse:[ri(3,par)]].
GEN> city2:[ifocus:dg_ifocus.owner:caller,reuse:[ri(3,neq)]].
GEN> caller:[owner:caller,reuse:[ri(3,eq)]].
GEN> discourserole2:[owner:system,reuse:[]].
GEN> Found Entry...
GEN> Gen from entry:eq, to#fw#
GEN> Gen from entry:par, london#n#
GEN> Gen from entry:par, leave#v#london#n#at#pr#
GEN> Gen from entry:noinst, want#v#to#fw#leave#v#london#n#at#pr#
GEN> Save reuse: reuse already saved for [semid>want1,reuse:mod,scope:lexical]_514981]
GEN> *******Surface tree generated
GEN>
GEN> h : h : morform : what
GEN>    lexical : qw
GEN> phrasal : ss
GEN> pfocus : scope : phrasal
GEN> ifocus : dg_ifocus
GEN> semid : s_time2
GEN> a : h : morform : time
GEN> lexical : n

316
The procedure describe_ids_ah searches in the accessibility history, finding subgraphs to span all the core ids, apart from those concerned with the carrier, for which a subgraph is built.

*****BEGINNING DESCRIPTION CYCLE*****
The subgraphs corresponding to location2 and s_time2 are spanned by the graph (root depart1) from entry 2). The remaining core graphs are attached to another graph (root arrive1) which is built parallel to this. The subgraph (root carrier1) gets attached. Finally, since the two parallel graphs are found to have material in common, a conjunction is built and returned.
****Building Graph...

*(sub)Graph [id:depart1,type:depart,...] spans subs [[id:s_time2,type:s_time,...],[id:location2,type:location,...]]

*Subs [[id:s_time1,type:s_time,...],[id:location1,type:location,...]] spanned by new graph

thejourney: id: single_journey
  type: single_journey
  thecarrier: id: carrier1
    type: carrier
    thecarrier_id: id: carrier_id1
      type: carrier_id
      value: ba
    thecarrier_number: id: number1
      type: number
      value: 123

id: arrive1
  type: arrive
  theplace: id: location1
    type: location
    thecity: id: city1
      type: city
      value: paris
  thetime: id: time_point1
    type: time_point
    thetime: id: s_time1
      type: s_time
      thehour: id: hour1
        type: hour
        value: 2
      theminutes: id: minutes1
        type: minutes
        value: 30

parallel to already built [id:depart1,type:depart,...]

GEN> *****INPUT SEMANTICS*******
GEN>
GEN> id: and1
GEN> type: and
GEN> first: id: depart1
GEN>   type: depart
GEN>   thetime: id: time_point2
GEN>     type: time_point
GEN>   thetime: id: s_time2
GEN>     type: s_time
GEN>   thehour: id: hour2
GEN>     type: hour
GEN>       t: id: gi(digit,1)
GEN>     type: digit
GEN>       value: 1
GEN>   u: id: gi(digit,2)
GEN>     type: digit
GEN>       value: 2
GEN>   theminutes: id: minutes2
GEN>     type: minutes
GEN>     t: id: gi(s_number,3)
GEN>     type: s_number
GEN>     value: 3
u : id : gi(s_number,0)
type : s_number
value : 0

thecity : id : city2
type : city
value : london

thecarrier_id : id : carrier_id1
type : carrier_id
value : 320

thecarrier_number : id : number1
type : number
value : 3

thecity : id : city1
type : city
value : paris

thetime : id : time_point1
type : time_point

thetime : id : s_time1
type : s_time

thetime : id : hour1
type : hour

thetime : id : minutes1
type : minutes

thecarrier_id : id : carrier_id1
GEN> type : carrier_id
GEN> 1 : id: gi(string,b)
GEN> type : string
GEN> value : b
GEN> 2 : id: gi(string,a)
GEN> type : string
GEN> value : a
GEN> thecarrier_number : id: number1
GEN> type : number
GEN> 1 : id: gi(digit,1)
GEN> type : digit
GEN> value : 1
GEN> 2 : id: gi(digit,2)
GEN> type : digit
GEN> value : 2
GEN> 3 : id: gi(digit,3)
GEN> type : digit
GEN> value : 3

GEN> ******ALIST******
GEN> and1: [owner: system, reuse: _615115].
GEN> depart1: [owner: caller, reuse: [ri(1, mod), ri(1, eq)]].
GEN> time_point2: [owner: caller, reuse: _615149].
GEN> s_time2: [ifocus: dg_ifocus, owner: caller, reuse: [ri(1, eq)]].
GEN> hour2: [ifocus: dg_ifocus, owner: caller, reuse: _615191].
GEN> minutes2: [ifocus: dg_ifocus, owner: caller, reuse: _615211].
GEN> location2: [ifocus: dg_ifocus, owner: caller, reuse: [ri(1, eq)]].
GEN> city2: [ifocus: dg_ifocus, owner: caller, reuse: _615253].
GEN> carrier1: [ifocus: dg_ifocus, owner: _608561, reuse: [new]].
GEN> carrier_id1: [ifocus: dg_ifocus, owner: _608642, reuse: [new]].
GEN> number1: [ifocus: dg_ifocus, owner: _608723, reuse: [new]].
GEN> arrive1: [owner: inferred(caller), reuse: [ri(O, par), ri(0, eq)]].
GEN> location1: [ifocus: dg_ifocus, owner: caller, reuse: [ri(0, neq)]].
GEN> city1: [ifocus: dg_ifocus, owner: caller, reuse: [ri(0, neq)]].
GEN> time_point1: [owner: caller, reuse: [ri(0, eq)]].
GEN> s_time1: [ifocus: dg_ifocus, owner: caller, reuse: [ri(0, par)]].
GEN> hour1: [ifocus: dg_ifocus, owner: caller, reuse: [ri(0, neq)]].
GEN> minutes1: [ifocus: dg_ifocus, owner: caller, reuse: [ri(0, valeq)]].
GEN> Found Entry...
GEN> Gen from entry: mod, leaves#v#london#n#at#par#twelve#n#thirty #npnpppli#
GEN> Save reuse: reuse already saved for [semid: depart1, reuse: mod, scope: lexical_764669]
GEN> Found Entry...
GEN> Found Entry...
GEN> Found Entry...
GEN> Gen from entry: par, in#par#paris#npp#
GEN> Gen from entry: par, at#par#two#n#thirty #npnppp#
GEN> Gen from entry: par, arrives#v#in#par#paris#npp#at#par#two#n#thirty #npnppp#
GEN> Save reuse: reuse already saved for [semid: arrive1, reuse: par, scope: lexical_568855]
GEN> ******Surface tree generated
GEN>
a : morform : one  
lexical : n  
phrasal : np  

pfocus : scope : lexical  
ifocus : dg_ifocus  
semid : gi(string,a)

a : morform : two  
lexical : n  
phrasal : np  

pfocus : scope : lexical  
ifocus : dg_ifocus  
semid : gi(digit,1)

a : morform : three  
lexical : n  
phrasal : np  

pfocus : scope : lexical  
ifocus : dg_ifocus  
semid : gi(digit,2)

h : a : h : h : h : morform : leave+s  
lexical : v  
phrasal : li  

pfocus : semid : depart1  
reuse : mod  
scope : lexical

a : h : h : morform : london  
lexical : n  

pfocus : reuse : eq  
scope : phrasal  
semid : city2  
ifocus : dg_ifocus

a : h : morform : at  
lexical : pr  
phrasal : pp  

pfocus : reuse : eq  
scope : phrasal

a : h : morform : twelve  
lexical : n  
phrasal : np  

pfocus : semid : hour2  
ifocus : dg_ifocus  
scope : phrasal

a : morform : thirty  
lexical : n  
phrasal : np  

pfocus : scope : lexical

h : morform : and  
lexical : cc  
phrasal : ss  

pfocus : semid : and1

a : h : h : h : morform : arrive+s  
lexical : v  

pfocus : semid : arrive1  
reuse : par  
scope : lexical

a : h : morform : in  
lexical : pr

322
<Illustrates: variation with constraint relaxation by Agent>

C: I want to travel from London to Paris: init; task; topic
A: What time do you want to arrive in Paris: init; query; open
C: Two thirty: resp; query; open
A: Two thirty: init; confirm; expl-
A: What time do you want to leave London: init; query; open
C: Leave London at twelve thirty: resp; query; open
A: ba123 leaves London at 1226 and arrives in Paris at 230: resp;query;topic

No core graphs can be successfully extracted from the accessibility history; this is a result of the opacity relation between the current view (world) and the previous views at which the earlier material was recorded. All core graphs are therefore built from scratch.

*****BEGINNING DESCRIPTION CYCLE*****

*** IDs to generate: [carrier_id1,carrier1,number1,location2,airport1,location1,airport2,city2,city1,time_point2,s_time2,hour2,minutes3,minutes1,s_time1,hour1]

*** AH Buffer:
1:
id : depart1
type : depart
thetime : id : time_point2
type : time_point
thetime : id : s_time2
type : s_time
thehour : id : hour2
type : hour
value : 12
theminutes : id : minutes2
type : minutes
value : 30

theplace : id : location2
type : location
thecity : id : city2
type : city
value : london

thejourney : id : single_journey1
type : single_journey
thepassenger : id : passenger1
type : passenger
theindividual : id : caller
type : individual
thediscoverrole : id : discoverrole

type : discoverrole
value : caller

2: [id:s_time2,type:s_time,thedesct:k:] 3: [id:s_time1,type:s_time,...]
4: backref(1)
5: [id:s_time1,type:s_time,thedesct:k:] 6: [id:go1,type:go,...] %2

*** Built Subgraphs: [id:carrier1

type : carrier
thecarrier_id : id : carrier_id1

type : carrier_id
value : ba

thecarrier_number : id : number1
type : number

324
value : 123

id : location2
type : location
theairport : id : airport1
type : airport
value : heathrow
thecity : id : city2
type : city
value : london

id : location1
type : location
theairport : id : airport2
type : airport
value : charles_de_gaulle
thecity : id : city1
type : city
value : paris

A graph parallel to entry 1) and another parallel to that are built, and conjoined as previously.

#####Building Graph ...
#####Subs [[id:time_point2,type:time_point,...] ,[id:s_time1,type:s_time,...] ] spanned by new graph
thejourney : id : single_journey1
type : single_journey
thecarrier : id : carrier1
type : carrier	hecarrier_id : id : carrier_id1
type : carrier_id
value : ba
thecarrier_number : id : number1
type : number
value : 123

id : depart1
type : depart
thelocation : id : location2
type : location
theairport : id : airport1
type : airport
value : heathrow
thecity : id : city2
type : city
value : london
thetime : id : time_point2
type : time_point
thetime : id : s_time2
type : s_time
thefhour : id : hour2
type : hour
value : 12
themintes : id : minutes3
type : minutes
value : 25

parallel to entry 1:[id:depart1,type:depart,...] %2
***Subs [[id:carrier1,type:carrier,...],[id:location1,type:location,...]]%2 , [id: s_time1,type:s_time,...] ] spanned by new graph

id : arrive1

type : arrive

thetime : id : time_point1

type : time_point

thetime : id : s_time1

type : s_time

thehour : id : hour1

type : hour

value : 2

theminutes : id : minutes1

type : minutes

value : 30

theplace : id : location1

type : location

theairport : id : airport2

type : airport

value : charles_de_gaulle

thecity : id : city1

type : city

value : paris

thejourney : id : single_journey1

type : single_journey

thecarrier : id : carrier1

type : carrier

thecarrier_id : id : carrier_id1

type : carrier_id

value : ba

thecarrier_number : id : number1

type : number

value : 123

parallel to already built [id:depart1,type:depart,...]]%2

GEN> ******INPUT SEMANTICS*******

GEN>

GEN> id : and1

GEN> type : and

GEN> first : id : depart1

GEN> type : depart

GEN> theplace : id : location2

GEN> type : location

GEN> theairport : id : airport1

GEN> type : airport

GEN> value : heathrow

GEN> thecity : id : city2

GEN> type : city

GEN> value : london

GEN> thetime : id : time_point2

GEN> type : time_point

GEN> thetime : id : s_time2

GEN> type : s_time

GEN> thehour : id : hour2

GEN> type : hour

GEN> t : id : gi(digit,1)

GEN> type : digit

GEN>

326
GEN> u : id : gi(digit,2)  
   type : digit  
   value : 2
GEN>  
GEN> theminutes : id : minutes3  
   type : minutes  
   t : id : gi(s_number,2)  
   type : s_number  
   value : 2
GEN>  
GEN> u : id : gi(s_number,5)  
   type : s_number  
   value : 5
GEN>
GEN> thetheme : id : carrier1  
   type : carrier  
   thecarrier_id : id : carrier_id1  
   type : carrier_id  
   1 : id : gi(string,b)  
      type : string  
      value : b
GEN>  
GEN> 2 : id : gi(string,a)  
      type : string  
      value : a
GEN>
GEN> thecarrier_number : id : number1  
   type : number  
   1 : id : gi(digit,1)  
      type : digit  
      value : 1
GEN>  
GEN> 2 : id : gi(digit,2)  
      type : digit  
      value : 2
GEN>  
GEN> 3 : id : gi(digit,3)  
      type : digit  
      value : 3
GEN>
GEN> rest : id : arrive1  
   type : arrive  
   thetime : id : time_point1  
   type : time_point  
   thetime : id : s_time1  
   type : s_time  
   thehour : id : hour1  
      type : hour  
      t : id : gi(digit,0)  
      type : digit  
      value : 0
GEN>  
GEN>  
GEN> u : id : gi(digit,2)  
      type : digit  
      value : 2
GEN>  
GEN> theminutes : id : minutes1  
   type : minutes  
   t : id : gi(s_number,3)  
   type : s_number  
   value : 3
GEN>  
GEN> u : id : gi(s_number,0)  
   type : s_number  
   value : 0
GEN>
GEN> theplace : id : location1  
   type : location  
   theairport : id : airport2  
   type : airport  
   value : charles_de_gaulle
thecity : id : city1
  type : city
  value : paris
thetheme : id : carrier1
  type : carrier
thecarrier_id : id : carrier_id1
  type : carrier_id
  1 : id : gi(string,b)
     type : string
     value : b
  2 : id : gi(string,a)
     type : string
     value : a
thecarrier_number : id : number1
  type : number
  1 : id : gi(digit,1)
     type : digit
     value : 1
  2 : id : gi(digit,2)
     type : digit
     value : 2
  3 : id : gi(digit,3)
     type : digit
     value : 3

******ALIST******
and1: [owner: system, reuse: _767311].
depart1: [owner: caller, reuse: [ri(1, mod), ri(1, eq)]].
city2: [ifocus: relaxation, owner: caller, reuse: _767392].
s_time2: [ifocus: relaxation, owner: caller, reuse: [ri(1, mod)]].
hour2: [ifocus: relaxation, owner: caller, reuse: [ri(1, valeq)]].
minutes3: [ifocus: system_relaxed, owner: caller, reuse: [ri(1, neq), ri(1, par)]].
carrier1: [ifocus: relaxation, owner: caller, reuse: [ri(1, mod), new]].
carrier_id1: [ifocus: relaxation, owner: caller, reuse: [ri(1, valeq), new]].
number1: [ifocus: relaxation, owner: caller, reuse: [ri(1, valeq), new]].
averri1: [owner: inferred(caller), reuse: [ri(1, par), ri(1, eq)]].
time_point1: [owner: caller, reuse: [ri(6, eq)]].
s_time1: [ifocus: relaxation, owner: caller, reuse: [ri(1, par)]].
hour1: [ifocus: relaxation, owner: caller, reuse: [ri(1, neq)]].
minutes1: [ifocus: relaxation, owner: caller, reuse: [ri(1, neq), ri(1, par)]].
location1: [ifocus: relaxation, owner: caller, reuse: [ri(1, par)]].
airport2: [ifocus: relaxation, owner: caller, reuse: [ri(1, neq)]].
city1: [ifocus: relaxation, owner: caller, reuse: [ri(1, neq)]].
gi(s_number, 2): [reuse: [ri(1, neq)]].
gi(s_number, 5): [reuse: [ri(1, neq)]]
gi(s_number, 3): [reuse: [ri(1, mod)]]
gi(s_number, 0): [reuse: [ri(1, mod)]]

Found Entry...
Found Entry...
Found Entry...
Gen from entry: noinst, heathrow np#
Found Entry...
Gen from entry: noinst, heathrow np#
Gen from entry: par, twenty#five npnnp#
Gen from entry: mod, at#pr#twelve#n#twenty#n#five npnnpnppp#
Gen from entry: mod, leaves#v#london#n#heathrow np#at#pr#twelve#n#twenty#n#five npnnpnpppli#
Save reuse: reuse already saved for [semid:time_point2, ifocus:relaxation, scope:phrasal, reuse:mod[644315]
Gen from entry:noinst, one #np#
Gen from entry:noinst, two #np#
Gen from entry:noinst, three #np#
Gen from entry:noinst, charles_de_gaulle #np#
Gen from entry:par, in#pr#paris#n#charles_de_gaulle #nppp#
Gen from entry:par, thirty #np#
Gen from entry:par, at#pr#two#n#thirty #npnppp#
Gen from entry:par, arrives#v#in#pr#paris#n#charles_de_gaulle #nppp#at#pr#two#n#thirty #npnppp#
Save reuse: reuse already saved for [semid:arrivel, reuse:par, scope:lexical|_774249]
*****Surface tree generated

h : a : h : h : h : h : morform : bee
lexical : n
phrasal : np
pfocus : ifocus : relaxation
semid : carrier1
a : morform : ay
lexical : n
phrasal : np
pfocus : scope : lexical
ifocus : relaxation
semid : gi(string,a)
a : morform : one
lexical : n
phrasal : np
pfocus : scope : lexical
ifocus : relaxation
semid : gi(digit,1)
a : morform : two
lexical : n
phrasal : np
pfocus : scope : lexical
ifocus : relaxation
semid : gi(digit,2)
a : morform : three
lexical : n
phrasal : np
pfocus : scope : lexical
ifocus : relaxation
semid : gi(digit,3)
h : a : h : h : h : morform : leave+s
lexical : v
phrasal : li
pfocus : semid : time_point2
ifocus : relaxation
scope : phrasal
reuse : mod
a : h : h : morform : london
lexical : n
GEN> pfocus : reuse : neq
GEN> scope : lexical
GEN> ifocus : relaxation
GEN> semid : city2
GEN> a : morform : heathrow
GEN> lexical : n
GEN> phrasal : np
GEN> pfocus : scope : phrasal
GEN> ifocus : relaxation
GEN> semid : airport1
GEN> a : h : morform : at
GEN> lexical : pr
GEN> phrasal : pp
GEN> pfocus : scope : lexical
GEN> a : h : morform : twelve
GEN> lexical : n
GEN> phrasal : np
GEN> pfocus : semid : hour2
GEN> ifocus : relaxation
GEN> scope : phrasal
GEN> reuse : mod
GEN> a : h : morform : twenty
GEN> lexical : n
GEN> phrasal : np
GEN> pfocus : semid : minutes3
GEN> ifocus : system_relaxed
GEN> reuse : par
GEN> scope : lexical
GEN> a : morform : five
GEN> lexical : n
GEN> phrasal : np
GEN> pfocus : reuse : neq
GEN> scope : lexical
GEN> ifocus : system_relaxed
GEN> semid : gi(s_number,5)
GEN> h : morform : and
GEN> lexical : cc
GEN> phrasal : ss
GEN> pfocus : semid : and1
GEN> a : h : h : h : morform : arrive+s
GEN> lexical : v
GEN> pfocus : semid : arrive1
GEN> reuse : par
GEN> scope : lexical
GEN> a : h : morform : in
GEN> lexical : pr
GEN> phrasal : pp
GEN> pfocus : scope : lexical
GEN> a : h : h : morform : paris
GEN> lexical : n
GEN> pfocus : semid : city1
GEN> ifocus : relaxation
GEN> scope : phrasal
GEN> reuse : par
GEN> a : morform : charles_de_gaulle
GEN> lexical : n
GEN> phrasal : np
GEN> pfocus : scope : phrasal
GEN> ifocus : relaxation

330
GEN> a : h : morform : at
GEN> lexical : pr
GEN> phrasal : pp
GEN> pfocus : scope : lexical
GEN> a : h : morform : two
GEN> lexical : n
GEN> phrasal : np
GEN> pfocus : semid : hour1
GEN> ifocus : relaxation
GEN> scope : phrasal
GEN> reuse : par
GEN> a : morform : thirty
GEN> lexical : n
GEN> phrasal : np
GEN> pfocus : semid : minutes1
GEN> ifocus : relaxation
GEN> scope : phrasal
GEN> reuse : par
GEN> ****Partial ordering:

GEN> [airport2-cityl,airport1-city2,carrierl-time_point2,minutes1-gi(digit,3),hour1-gi(digit,3),city1-gi(digit,3),gi(s_number,5)-gi(digit,3),minutes3-gi(digit,3),minutes1-gi(digit,2),hour1-gi(digit,2),city1-gi(digit,2),gi(s_number,5)-gi(digit,2),minutes3-gi(digit,2),minutes1-gi(digit,1),hour1-gi(digit,1),city1-gi(digit,1),gi(s_number,5)-gi(digit,1),minutes3-gi(digit,1),minutes1-gi(string,a),hour1-gi(string,a),city1-gi(string,a),gi(s_number,5)-gi(string,a),minutes3-gi(string,a),minutes1-hour2,hour1-hour2,city1-hour2,gi(s_number,5)-hour2,minutes3-hour2,minutes1-airport1,hour1-airport1,city1-airport1,gi(s_number,5)-airport1,minutes3-airport1,gi(digit,3)-carrier1,gi(digit,2)-carrier1,gi(digit,1)-carrier1,gi(string,a)-carrier1,hour2-carrier1,city2-carrier1,arrive1-and1,time_point2-arrive1]
GEN> Totally ordered ids
GEN> and1:0.
GEN> arrive1:1.
GEN> time_point2:2.
GEN> carrier1:3.
GEN> gi(digit,3):4.
GEN> gi(digit,2):4.
GEN> gi(digit,1):4.
GEN> gi(string,a):4.
GEN> hour2:4.
GEN> city2:4.
GEN> city1:5.
GEN> airport1:5.
GEN> airport2:6.
GEN> minutes1:6.
GEN> hour1:6.
GEN> city1:6.
GEN> gi(s_number,5):6.
GEN> minutes3:6.
GEN> #See#one#two#three#leaves#at#london#heathrow#at#pr#twelve#twenty#five:#paris#charlesdegaule#np#at#pr#two#np#thirty#np#paris.
The following illustrates the generation of an echo corresponding to two task parameters.

*****BEGINNING DESCRIPTION CYCLE******

*** IDs to generate: [city2, location2, city1, location1]

*** AH Buffer:

```
1:
id : go1
type : go
thegoal : id : location1
 type : location
 thecity : id : city1
 type : city
 value : paris
thesource : id : location2
 type : location
 thecity : id : city2
 type : city
 value : luton
thetheme : id : caller
 type : individual
 thediscourserole : id : discourserole1
 type : discourserole
 value : hearer
```

*** Found Subgraphs: [[id:location1,type:location,...],[id:location2,type:location,...]]

****Building Graph ...

**(sub)Graph

```
id : go1
type : go
thegoal : id : location1
 type : location
 thecity : id : city1
 type : city
 value : paris
thesource : id : location2
 type : location
 thecity : id : city2
 type : city
 value : luton
thejourney : id : single_journey1
 type : single_journey
 thepassenger : id : passenger1
 type : passenger
 theindividual : id : caller
 type : individual
 thediscourserole : id : discourserole
```

1
type : discourserole
value : hearer

from AH entry 1 spans subs [[id:location1,type:location,...],[id:location2,type:location,...]]

Gen from entry: eq, travel#v#to#pr#paris#npp#from#pr#luton#npp#
**Generation proceeds similarly to questions about time (cf NFENQ111) and is not shown.**

"C: not Luton---London: init; correct
"A: you want to travel from LONDON: resp; correct

The core graph corresponding to London is capable of being used, since the system accepts the view (world index) corresponding to the caller's modification

****BEGINNING DESCRIPTION CYCLE*******

**** IDs to generate: [location2, city3]

**** AH Buffer:

1:
id : location2
type : location
type : location
thecity : id : city3
type : city
value : london

2:
id : location2
type : location
thecity : id : city2
type : city
value : luton
modus : pol : neg

3: [id:s_date1,type:s_date,thedes:...
4: [id:go1,type:go,...
5: backref(1,)

*** Found Subgraphs: [[id:location2,type:location,...]%2 ]
*****Building Graph ...
*****Built new graph:

thef Journey : id : single_journey1
type : single_journey
thepassenger : id : passenger1
type : passenger
theindividual : id : caller
type : individual
thediscourserole : id : discourserole

type : discourserole

id : go1

type : go
thecity : id : city3
type : city
value : london
to cover subs [[id:location2,type:location,...]%2 ]

***Accessibility information: go1:[owner:inferred(caller),reuse:].
location2:[ifocus:dg_ifocus,owner:caller,reuse:[ri(1,eq)]].
city3:[ifocus:caller_authd,owner:caller,reuse:].
caller:[owner:caller,reuse:[ri(3,eq)]].
discourserole2:[owner:system,reuse:].

GEN> *****INPUT SEMANTICS******
GEN> id : want1
type : want
thecagent : id : caller
type : individual
thediscourserole : id : discourserole2
type : discourserole
value : hearer
thetheme : id : go1
type : go
thecsource : id : location2
type : location

**Surface tree generated**

```
GEN> Gen from entry: eq, you #op#
GEN> Gen from entry: eq, from#pr#london#npp#
GEN> Gen from entry: noinst, travel#v#from#pr#london#npp#
GEN> Gen from entry: eq, from#pr#london#npp#
GEN> *****Partial ordering:
GEN> city3 /[ go1 /[ want1 /[ caller /[ ]]]].
GEN> Totally ordered ids
gen> caller: 0.
```

336
want:1.
gol:2.
city3:3.
#1: you #1: want #1: to #1: travel #1: from #1: London #1: to #1: Paris #1: .

Illustrates: Agent corrects Caller

C: i want to travel from London to Paris: init; task; topic
A: what time do you want to arrive in Paris: init; query; open
C: two thirty: resp; query; open
A: two thirty: init; confirm; explain
A: what time do you want to leave London: init; query; open
C: leave London at twelve thirty: resp; query; open
A: ba123 leaves London at 1230 and arrives in Paris at 230: resp; query; topic
C: ba223
A: ba123

The previous (correct) subgraph corresponding to ba123 cannot be used; if it were, then it could be treated as an echo. It is blocked by the more recent presence of a modification. The subgraph is therefore built anew.

*****BEGINNING DESCRIPTION CYCLE*****

*** IDs to generate: [number1, carrier1, carrier_id1]
*** AH Buffer:
  1: id : carrier1
type : carrier
thecarrier_id : id : carrier_id1
type : carrier_id
value : ba
thecarrier_number : id : number2
type : number
value : 223

  2: [id:and1, type: and, ...]
  3: [id:depart1, type: depart, ...] 2
  4: [id: s_time2, type: s_time, thedesc: ...]
  5: backref(3, [rest, thetime, thetime])
  6: backref(1,)
  7: [id: s_time1, type: s_time, thedesc: ...]

*** Built Subgraphs: [[id: carrier1, type: carrier, ...]]
*****Building Graph ...
***Subs [[id: carrier1, type: carrier, ...]] spanned by new graph [id: carrier1, type: carrier, ...]

parallel to entry 1: [id: carrier1, type: carrier, ...] 2
GEN>
  type : string
GEN>
  value : a
GEN> thecarrier_number : id : number1
GEN>
  type : number
GEN> 1 : id : gi(digit,1)
GEN>
  type : digit
GEN>
  value : 1
GEN> 2 : id : gi(digit,2)
GEN>
  type : digit
GEN>
  value : 2
GEN> 3 : id : gi(digit,3)
GEN>
  type : digit
GEN>
  value : 3
GEN> ******ALIST******
GEN> carrier1 : [ifocus : dg_ifocus, owner : caller, reuse : [ri(1,mod)]]
GEN> carrier_id1 : [ifocus : dg_ifocus, owner : caller, reuse : [ri(1,valeq)]]
GEN> number1 : [ifocus : system_authd, owner : caller, reuse : [ri(1,neq)]]
GEN> gi(digit,1) : [reuse : [ri(1,seq)]]
GEN> gi(digit,2) : [reuse : [ri(1,mod)]]
GEN> gi(digit,3) : [reuse : [ri(1,mod)]]
GEN> Found Entry...
GEN> Gen from entry : eq, ay#n#
GEN> Found Entry...
GEN> Gen from entry : eq, two#n#
GEN> Gen from entry : mod, three#n#
GEN> Gen from entry : mod, bee#n#ay#n#one#n#two#n#three#nnp#
GEN> Save reuse : reuse already saved for [semid : carrier1, ifocus : dg_ifocus, reuse : mod, scope : lexical]_909888]
GEN> ******Surface tree generated
GEN>
GEN>
GEN> h : h : h : h : morform : bee
GEN>
  lexical : n
GEN>
  phrasal : np
GEN> Found Entry...
GEN> Gen from entry : eq, ay #np#
GEN> Found Entry...
GEN> Gen from entry : eq, two #np#
GEN> Gen from entry : mod, three #np#
GEN> Gen from entry : mod, bee#n#ay #np#one #np#two #np#three #npnp#
GEN> Save reuse : reuse already saved for [semid : carrier1, ifocus : dg_ifocus, reuse : mod, scope : lexical]_831738]
GEN> ******Surface tree generated
GEN>
GEN>
a : morform : one
lexical : n
phrasal : np
pfocus : reuse : neq
scope : lexical
ifocus : system_authd
semid : gi(digit,1)

a : morform : two
lexical : n
phrasal : np
pfocus : semid : gi(digit,2)
reuse : mod
scope : phrasal
ifocus : dg_ifocus

a : morform : three
lexical : n
phrasal : np
pfocus : semid : gi(digit,3)
reuse : mod
scope : lexical
ifocus : dg_ifocus

**Partial ordering:**
gi(digit,1) / [carrier1 /

gi(digit,2) /

gi(digit,3) /

gi(string,a) /]

Totally ordered ids

gi(digit,3):1.
gi(digit,2):1.
carrier1:1.
gi(digit,1):2.

#1: #bee#n: | | : #ay #: #one #: enp# : #two #: np# : #three #: npnp#.

---

Illustrates: Agent corrects Caller;
additional information (airports);
rhetorical focus applied

C: i want to travel from London to Paris: init; task; topic

A: what time do you want to arrive in Paris: init; query; open

C: two thirty: resp; query; open

A: two thirty: init; confirm; expl-

A: what time do you want to leave London: init; query; open

C: leave London at twelve thirty: resp; query; open

A: ba123 leaves London at 730 and arrives in Paris at 827: resp; query; topic

C: 817

A: it arrives in r: paris charles de gaulle : r at eight TWENTY seven

msg(bm,mp, request_description_object(mp10,know(rh_depend, avs(dbflight1, [goalcity: paris, goalairport: charles_de_gaulle, goaltime: 827]), carrier))).
The description request requires that the goal time be re-stated, together with the goal city and the (new) goal airport. The carrier is not explicitly mentioned; the final argument however will force the theme to be of type carrier, rather than the caller.

*****BEGINNING DESCRIPTION CYCLE*****

*** IDs to generate: [city1,airport1,location1,minutes1,s_time1,hour1]
*** AH Buffer:
1: [id:s_time1,type:s_time, ...]
2: [id:and1,type:and, ...]
3: [id:depart1,type:depart, ...] \%2
4: [id:s_time2,type:s_time, thedesc: ...]
5: backref(3,[rest, thetime, thedesc])
6: backref(1,)
7: [id:s_time1,type:s_time, thedesc: ...]

*** Found Subgraphs: [[id:location1,type:location, ...] ]
*** Built Subgraphs: [[id:s_time1,type:s_time, ...] , id : airport1
   type : airport
   value : charles_de_gaulle ]

*** Building Graph ...
*** Subs [[id:location1,type:location, ...] , [id:s_time1,type:s_time, ...] ] spanned by new graph

thejourney : id : single_journey1
   type : single_journey
   thecarrier : id : carrier1
      type : carrier
      thecarrier_id : id : carrier_id1
         type : carrier_id
         value : ba
      thecarrier_number : id : number1
         type : number
         value : 123

id : arrive1
   type : arrive
   thetime : id : time_point1
      type : time_point
      thetime : id : s_time1
         type : s_time
         thehour : id : hour1
            type : hour
            value : 8
         theminutes : id : minutes1
            type : minutes
            value : 27

theplace : id : location1
   type : location
   thecity : id : city1
      type : city
      value : paris

parallel to entry 2:[id:arrive1,type:arrive,...] \%2

A simple heuristic is used to introduce a pronominalisation flag for carrier1.
**INPUT SEMANTICS**

```plaintext
id : arrival

thetime : id : time_point

dtype : time_point

dthetime : id : s_time

dtype : s_time

dthehour : id : hour

dtype : hour
dt : id : gi(digit,0)
dtype : digit
dvalue : 0
du : id : gi(digit,8)
dtype : digit
dvalue : 8

dtheminutes : id : minutes

dtype : minutes
dt : id : gi(s_number,2)
dtype : s_number
dvalue : 2
du : id : gi(s_number,7)
dtype : s_number
dvalue : 7

dtheplace : theairport : id : airport

dtype : airport
dvalue : charles_de_gaulle

did : location

dtype : location

dthecity : id : city

dtype : city
dvalue : paris

dthetheme : id : carrier

dtype : carrier

dmodus : ref : pro
```

**ALIST**

```plaintext
s_time1:[rh_fun(location1,rn),ifocus:dg_ifocus,owner:caller, reuse:[ri(2,mod ),ri(1,mod )]].

dlocation1:[rh_fun(s_time1,dom),ifocus:dg_ifocus,owner:caller, reuse:[ri(2,mod )]].

darrival1:[owner:inferred(caller),reuse:[ri(2,mod),ri(2,eq)]]

dtime_point1:[owner:caller, reuse:[ri(2,eq)]]

dhour1:[ifocus:dg_ifocus,owner:caller, reuse:[ri(2,valeq),ri(1,valeq)]]

dminutes1:[ifocus:system_authd, owner:caller, reuse:[ri(2,valeq),ri(1,neq),ri(1,par)]]

dairport1:[ifocus:dg_ifocus,owner:caller, reuse:[new]].
dcity1:[ifocus:dg_ifocus,owner:caller, reuse:[ri(2,valeq)]]
dcarrier1:[owner:caller, reuse:[ri(2,eq)]]
gi(s_number,2):[reuse:[ri(1,neq)]]
gi(s_number,7):[reuse:[ri(1,mod)]]

Found Entry...

Found Entry...

Found Entry...

Gen from entry:noinst, charles_de_gaulle #np#

Gen from entry:mod, in#prr:#paris#n:rr:#charles_de_gaulle #:rnppp#

Found Entry...
```

341
Gen from entry: mod, seven #np#
Gen from entry: par, twenty#n#seven #np#np#
Gen from entry: mod, at#pr#eight#n#twenty#n#seven #np#np#np#
Gen from entry: mod, it #np#arrives#v#in#prr:#paris#n:rr:#charles_de_gaulle
#:rnp#np#np#np#at#pr#eight#n#twenty#n#seven #np#np#np#np#
Gen> Save reuse: reuse already saved for [semid: arrive1, reuse: mod, scope: lexical|_1087144]
GEN> pfocus : semid : gi(s_number,7)
GEN> reuse : mod
GEN> scope : lexical

GEN> ****Partial ordering:
GEN> [airport1-city1,gi(s_number,7)-arrive1,minutes1-hour1,minutes1-airport1,arrive1-carrier1,hour1-gi(s_number,7),city1-gi(s_number,7)]
GEN> Totally ordered ids
GEN> carrier1:0.
GEN> arrive1:1.
GEN> gi(s_number,7):2.
GEN> hour1:3.
GEN> city1:3.
GEN> airport1:4.
GEN> minutes1:5.
GEN> %]:=it #:np|#:arrives\#v:\#in\#prr\#:paris\#n:rr\#:charlesdegaule \#:rnppl\#at\#p r\#:eight\#n:tw\#:twenty\#n\#:seven \#n\#npn\#pp\#s\#.

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
 << S4
 << Illustrates: Caller overrides Agent default
 <<
 << C: when is the next flight to Stuttgart: init;query;open;topic
 << A: you want to travel from London Heathrow: init;query;default

*****BEGINNING DESCRIPTION CYCLE*****

*** IDs to generate: [location1,city1]
*** AH Buffer:
1: [id:s_time1,type:s_time,thesc:...]

*** Found Subgraphs: [[id:location1,type:location,...] ]
*** Building Graph ...
*** (sub)Graph [id:location1,type:location,...] from AH entry 1 spans subs [[id:location1,type:location,...] ]
*** Accessibility information: location1:[ifocus:dg_ifocus,owner:caller,reuse:[ri(1,eq)].
    city1:[ifocus:dg_ifocus,owner:caller,reuse:].

GEN> *****Surface tree generated
GEN>
GEN> h : h : a : morform : you
GEN> lexical : n
GEN> phrasal : op
GEN> pfocus : semid : individual1
GEN> h : morform : want
GEN> lexical : v
GEN> phrasal : ss
GEN> pfocus : semid : want1
GEN> a : morform : to
GEN> lexical : fw
GEN> a : h : h : h : h : morform : travel
GEN> lexical : v
GEN> a : pfocus : semid : go1
GEN> a : h : morform : from
LEXICAL

**BEGINNING DESCRIPTION CYCLE**

***IDs to generate: [airport2, location2]***

***AH Buffer:***

1: [id: location2, type: location, ...]
2: [id: go1, type: go, ...]
3: [id: location1, type: location, ...]
4: [id: s_time1, type: s_time, thedesc: ...]

***Found Subgraphs: [[id: location2, type: location, ...]]***

***Building Graph ...***

**Built new graph:**

thejourney : id : single_journey1
type : single_journey
thepassenger : id : passenger1
type : passenger
theindividual : id : individual1
type : individual
id : id : caller
type : individual
thediscoverrole : id : discours
erole2

type : discou
rvalue : heare

id : go1

type : go

thesource : id : location2
type : location
th.airport : id : airport2
type : airport
value : stansted
to cover subs [[id:location2,type:location,... ]]

***Accessibility information: go1:[owner:inferred(caller),reuse:].
location2:[ifocus:dg_ifocus,owner:caller,reuse:[ri(1,eq)]].
airport2:[ifocus:override_default,owner:caller,reuse:].
individual1:[owner:caller,reuse:[ri(2,eq)]].

GEN> *****Surface tree generated
GEN>
GEN> h : h : a : morform : you
GEN> lexical : n
GEN> phrasal : op
GEN> pfocus : semid : individual1
GEN> h : morform : want
GEN> lexical : v
GEN> phrasal : ss
GEN> pfocus : semid : want1
GEN> a : morform : to
GEN> lexical : fw
GEN> a : h : h : h : h : morform : travel
GEN> lexical : v
GEN> pfocus : semid : go1
GEN> a : h : morform : from
GEN> lexical : pr
GEN> phrasal : pp
GEN> pfocus : ifocus : dg_ifocus
GEN> a : h : h : morform : london
GEN> lexical : n
GEN> pfocus : ifocus : dg_ifocus
GEN> semid : location2
GEN> a : morform : stansted
GEN> lexical : n
GEN> phrasal : np
GEN> pfocus : scope : phrasal
GEN> ifocus : override_default
GEN> semid : airport2

GEN> *****Partial ordering:
GEN> airport2 /[
GEN> location2 /[
GEN> go1 /[
GEN> want1 /[
GEN> individual1 /[]].
GEN>
GEN> Totally ordered ids
GEN> individual1:0.
GEN> want1:1.
GEN> go1:2.
GEN> location2:3.
GEN> airport2:4.
GEN> #|:#you #:|op|:#want#v:#|to#fw#travel#v#|from#pr5#london#n6#stansted #nppps #.

345