# Chapter 11

# Sentence Generation

## 1. What is Sentence Generation?

The aim of a typical text generation system is to produce a text which satisfies some set of pre-stated goals. Such systems are provided with a knowledge base -- which contains information to be expressed -- and a set of goals. The system then organises this information into sentence-length chunks, realises these chunks as sentences, and prints or speaks the text. Figure 11.1 shows a typical application of a text generation system: a weather satellite beams down weather information to a receiver dish, which passes the information to a computer. The computer draws upon this knowledge base (and perhaps other sources) to generate a weather report.

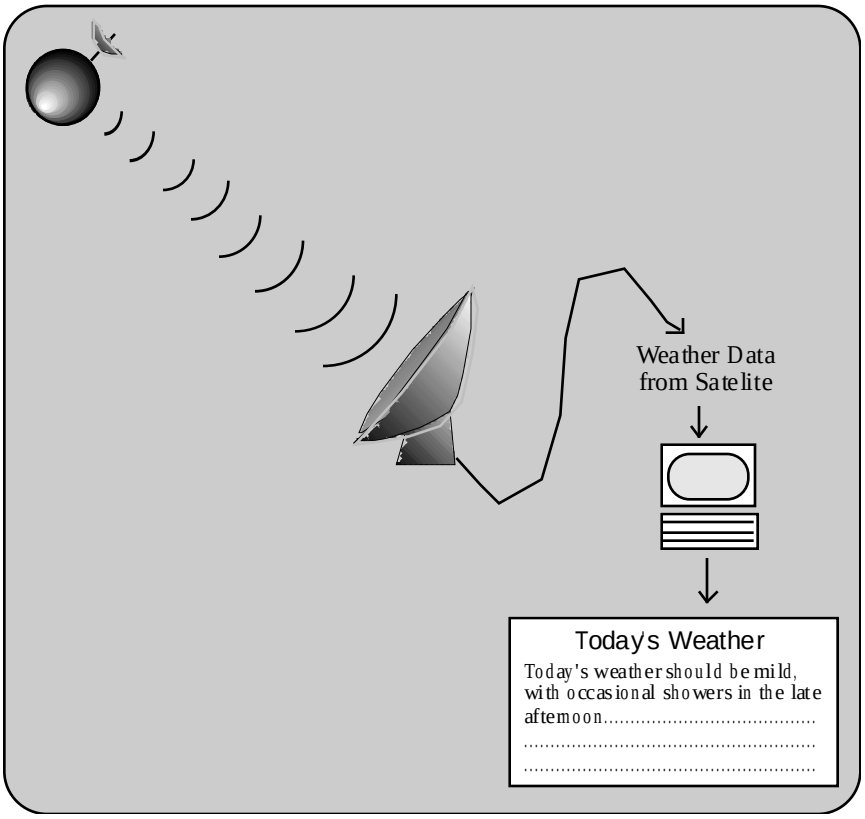


Figure 11.1: A Weather Report Generation System

A possible architecture for this text generation process is shown in figure 11.2, and the three stages (which can be inter-mixed) are described below.

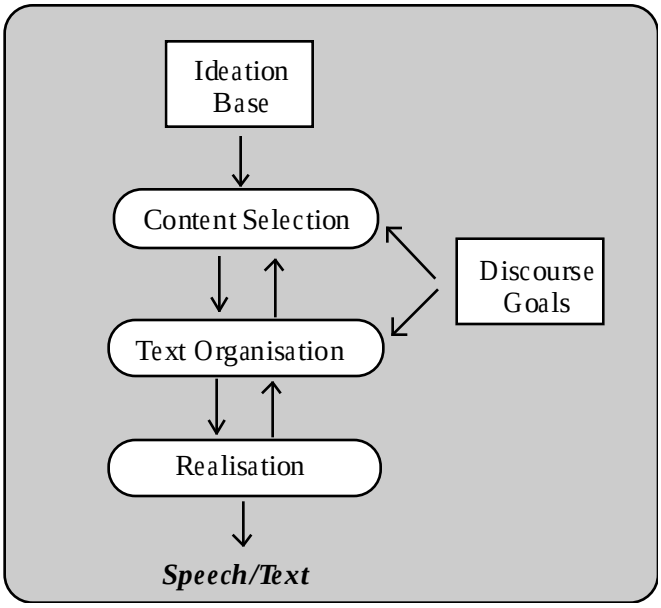


Figure 11.2: From Knowledge-base to Text/Speech

1) **Content Selection**: Determining which of the facts in the ideational (knowledge) base need to be expressed to best achieve the discourse goals.
2) **Text Organisation**: Splitting the selected content into segments realisable as single sentences (micro-semantic specifications), and ordering these segments into a sequence which best achieves the discourse goals. Different discourse goals may result in different orderings.
3) **Realisation**: realising these segments as sentences.

WAG provides only the sentence realisation component of a multi-sentential text generation system. I will thus focus on how micro-semantic specifications are realised as graphological strings[1]. This chapter provides firstly a general background to sentence generation (general issues), and then a detailed description of the WAG generator.

Text generation is a relative new-comer to the NLP field: while some papers were published in the late sixties and seventies (e.g., Friedman 1969; Simmons & Slocum 1972), generation did not really take off until the eighties. As a field, it has been relatively small compared to the analysis field. However, the focus on generation is growing, since it is important for machine translation and natural language interfaces.

Sentence generation using Systemic grammar have been prominent throughout the development of text generation, including Proteus (Davey, 1974/1978), Penman (Mann 1983a; Mann & Matthiessen 1985) and Slang (Patten 1986, 1988). In recent years, several other Systemic generation systems have been implemented, including Genesys (Fawcett & Tucker 1990) and Horace (Cross 1991).

Important non-Systemic sentence generators include Mumble (McDonald 1980; Meteer *et al.* 1987); KAMP (Appelt 1985); TAG (Joshi & Vijay-shankar 1985); that of Danlos (Danlos 1984); and FUF (Elhadad 1991).

[1] As stated in chapter 2, we are focusing on graphological rather than speech output. Speech can be produced by the WAG system, but this is through a text-to-speech module connected to the program.

# 2. Some Issues in Sentence Generation

This section discusses some methodological issues in the construction of a sentence generation system.

## 2.1 Flow of Information

In chapter 7, I discussed three information-flow architectures: conduit (staged), interleaved, and integrated.

The popularity of the interleaved approach is growing in the generation area, since decisions about *what to say* (content selection module) depend on the choice of *how to say* (realisation module). For example, when expressing a process, the decision to include the Actor and Actee will depend on whether a clause or nominal-group is used to express the process -- a nominal expression including both Actor and Actee is more awkward, and thus less likely, e.g., *the buying of the book by Mary*.

Integrated architectures are also gaining popularity. As formalisms for representing natural language resources become more robust, many systems are now using the same language to represent all levels of representation: semantic structures, text-structures, and lexico-grammar. Loom, for instance, can be used to represent rhetorical structure trees (Hovy *et al.* 1992), ideational structures (Bateman *et al.* 1990), and lexico-grammatical structures (Kasper 1989). Once all the resources are in the same formalism, the processor can ignore the stratal division of the resources, and perform one operation only -- propagating the logical implications of the input, thus deriving the output structure.

## 2.2 Control Strategies

In chapter 7, I proposed that most NLP can be viewed as a process of translating between strata: building a target representation based on a source representation. Control strategies handle the mapping between any two representational levels. In a tri-stratal system, we need two control strategies (assuming a conduit architecture): one between micro-semantics and lexico-grammar; and one between lexico-grammar and graphology.

### 2.2.1 Between Micro-Semantics and Lexico-Grammar

To produce a lexico-grammatical representation from a micro-semantic representation, we can use either a source-driven (data-directed), or a target-driven (goal-directed) strategy:

1) **Source-Driven**: each feature of the micro-semantic specification has associated lexico-grammatical constraints (the lexico-grammatical consequences of the semantic feature). To build a lexico-grammatical structure, we take each feature of the micro-semantic specification in turn, and apply its lexico-grammatical realisations. This is repeated for each unit in the structure. In this way we build up a lexico-grammatical structure.
2) **Target-Driven**: the inter-stratal mapping constraints are represented as semantic constraints on lexico-grammatical features. To build a lexico-grammatical structure that encodes the micro-semantic input, we traverse the lexico-grammatical network, choosing a feature in each system whose semantic constraint matches the micro-semantic specification. Basically, we build a lexico-grammatical structure in the grammar's own terms, although the choices are constrained by the semantics.

Most of the Systemic generation systems use the target-driven approach (e.g., Penman, Proteus, Genesys). The following quote from Matthiessen (1985) demonstrates this for the Penman system:

"In Nigel ... initiative comes from the grammar, the general control of what happens comes from the entry conditions of the systems. It is not the case that the semantic stratum has its own control, does its work and presents the results to the grammar for realisation. Instead, it is controlled by the entry conditions of the systems."

Patten's SLANG system is the exception: grammatical features are preselected as the realisations of the semantic features:

"Features at the semantic stratum may have realisation rules which preselect grammatical features. Similarly, grammatical features may preselect features from the phonological/orthographic stratum." (1988, p44).

We can also distinguish between resource-driven and representation-driven systems: a resource-driven system uses the resources to select the next rule or constraint to apply, while a representation-driven system uses the information in the representation to control the structure-building. Penman uses a mixture of both -- firstly, it is representation-driven to the extent that the unit-of-focus -- the element being expanded -- is chosen in reference to the lexico-grammatical representation: we start with the top element (the clause), and successively expand elements down towards the leaves of the tree. This represents a top-down, breadth-first generation strategy. However, within each unit, the construction is resource-driven: the system network is used to control the construction of each unit's internal structure. The unit is constructed by a forward-traversal through the network, asserting the realisations of each feature selected. In simpler terms, the node-selection strategy is representation-driven, and the rule-selection strategy is resource-driven.

The WAG generator follows the Penman tradition, using the lexico-grammar to control the generation, expanding units in a top-down, breadth-first manner. Each unit is constructed as a result of a traversal of the system network. Section 4.3 below will discuss the strategy in more detail.

### 2.2.2 Between Lexico-grammar and Graphology

While Penman and WAG both use a target-driven control strategy between micro-semantic and lexico-grammar, in the graphological construction, control is source-driven. The source, in this case, is the lexico-grammatical structure. The process finds the leaves of this structure (word-rank elements), and recovers the associated lexical-items. Using these items, and inflectional features, the appropriate graphological-forms are generated, and printed (with formatting, e.g., capitalisation, spacing, etc.). Graphological generation in the WAG system will be discussed more fully in section 4.5 below.

## 2.3 Deterministic *vs.* Non-Deterministic Generation

In chapter 9, I described the issue of determinism in parsing - whether the parser resolves each choice before continuing (deterministic parsing), or whether it explores each alternative (non-deterministic parsing).

These same possibilities apply for generation also. We may reach a point in the generation process where two alternative means of expressing the semantics both seem valid. Often, each of the choices will lead to appropriately generated sentences, the choices representing alternative means of realising the meaning.[2] In other cases, however, some choices may lead to a dead-end in the generation process -- no appropriate realisation is possible. This has been called a *generation gap* (Meteer 1990).

---

[2] In a fully-constrained system, all differences in form would be linked back to differences in meaning. However, at present, it is difficult to assign meaning differences to all form differences, e.g., the semantic difference between "I said that he was coming" and "I said he was coming". Such differences are defaulted in the WAG system.

Generation gaps occur because choices are often dependent on each other -- if we make the wrong choice at one point, there may be no valid alternatives at a later choice-point. For instance, Meteer (1990, p63) gives an example of the generation of a process involving someone deciding something important. At one point in the generation, we face a choice between congruent realisation -- *He decided* -- or an incongruent realisation -- *He made a decision.* Both choices seem equally valid. However, the incongruent choice allows the 'important' characteristic to be expressed -- *He made an important decision* -- while the congruent choice does not -- **He decided importantly*.

When the decisions on which a particular choice depends are not made *before* the choice is reached, then we have a determination problem -- the choice cannot be resolved. I will discuss below the two types of solution to this problem -- forcing a decision (deterministic generation), and following all alternatives (non-deterministic generation).

### 2.3.1 Non-Deterministic Generation

A non-deterministic generator doesn't make a definite decision between alternatives, but either chooses one tentatively, or follows all alternatives simultaneously. The same strategies that are available for non-deterministic parsing (see chapter 9) are also available for generation:

- **Simultaneous Generation**: all options are carried forward at the same time. This option includes 'chart generation', along the lines of chart parsing (cf. Haruno *et al.* 1993).
- **Backtracking Generation**: at each choice-point, an arbitrary decision is made. When a generation dead-end is reached, the generator backtracks to the last choice-point, makes a different choice, and proceeds from there. At one stage I modified the WAG generator to allow backtracking. However, this generation was very inefficient due to the large size of the backtracking stack which needed to be saved. For this reason I have switched to deterministic generation[3].

### 2.3.2 Deterministic Generation

In deterministic generation, the process resolves choices as they are reached. A problem for this approach is that there is not always sufficient information to make the decision available.

Matthiessen (1988a) points out one problem-case for non-deterministic Systemic generation: "How is the situation to be avoided where a chooser is entered before all the hub associations needed are in place?" (p775). In terms of WAG, this problem is stated as follows: the generator wishes to test a feature selection-constraint which includes a reference to the Referent role of some unit. However, the filler of the Referent role has not yet been established. The establishment of the Referent role is performed in some other system, which has not yet been entered. The feature selection-constraint thus cannot be tested.

This kind of problem is common in writing Systemic grammars of reasonable complexity. For instance, when choosing between the features *single-subject* and *plural-subject* (concerning Subject-Finite agreement), the selection-constraints refer to *Subject.Referent*, but the Subject's *Referent* role may not have been established yet. It is established in a simultaneous system, where the *Subject* is conflated with either the *Agent*, *Medium* or *Beneficiary*.

---

[3] A variation of this approach stores only the choice made at each decision point, and not the generation environment. When generation fails, the process goes back to the beginning of the generation and re-creates the structure, varying only the last choice. This approach has the advantage of far less storage space requirements. However, the same structure-building work might be done over and over, meaning that this approach will be slow if any degree of backtracking occurs. The approach is appropriate if the number of backtracks is assumed to be very small, e.g., the first path is likely to succeed, but we allow for the possibility of failure.

Nigel and WAG have avoided such non-deterministic problems, by careful writing of the lexico-grammatical and interstratal resources. However, this is a case where the resources are being shaped by the needs of the process, a practice which should be avoided, if possible, since the resources lose their process-neutrality.

Matthiessen (1988a) proposed one solution which avoids the re-wiring of the grammar. He proposes a *least-commitment strategy* -- whenever a grammatical choice cannot be resolved, then we should make no commitment, but rather postpone the decision until a later point. There are potentially other grammatical decisions which can be made without waiting for this one (e.g., simultaneous systems). The system is pushed to the end of the systems-to-be-resolved queue.

This is a good solution for some cases, since it doesn't require any change to the resources -- only the traversal algorithm is affected. However, the solution cannot be used in two situations:

1) **Inter-Dependency**: there may be cases where two decisions depend on each other. Each decision cannot be resolved until the other decision is resolved.
2) **Referent resolved in more delicate system**: sometimes the Referent is resolved in a more delicate system, rather than in a simultaneous system. No amount of delay will solve the problem.

We could write the resources to avoid these situations (while allowing cases which could be solved using least-commitment). Alternatively, we could introduce more complex processes which know how to obtain as needed the information required to resolve the choices. I will not discuss this further here, except to say that the concept of 'look-ahead' from parsing could perhaps be applied profitably.

## 3. WAG's Input Specification

This section and the next describe WAG's sentence generation system. I start by describing the input to the system and then describe the generation algorithm itself.

The input to the generation process is a micro-semantic representation[4]. Figure 11.3 shows a sample micro-semantic specification, from which the generator would produce: "I'd like information on some panel beaters.". The distinct contributions of the three meta-functions are separated by the grey boxes.

[4]WAG can also be set to generate without semantic constraint, by making random or default lexico-grammatical selections, or by allowing a human to make these decisions.

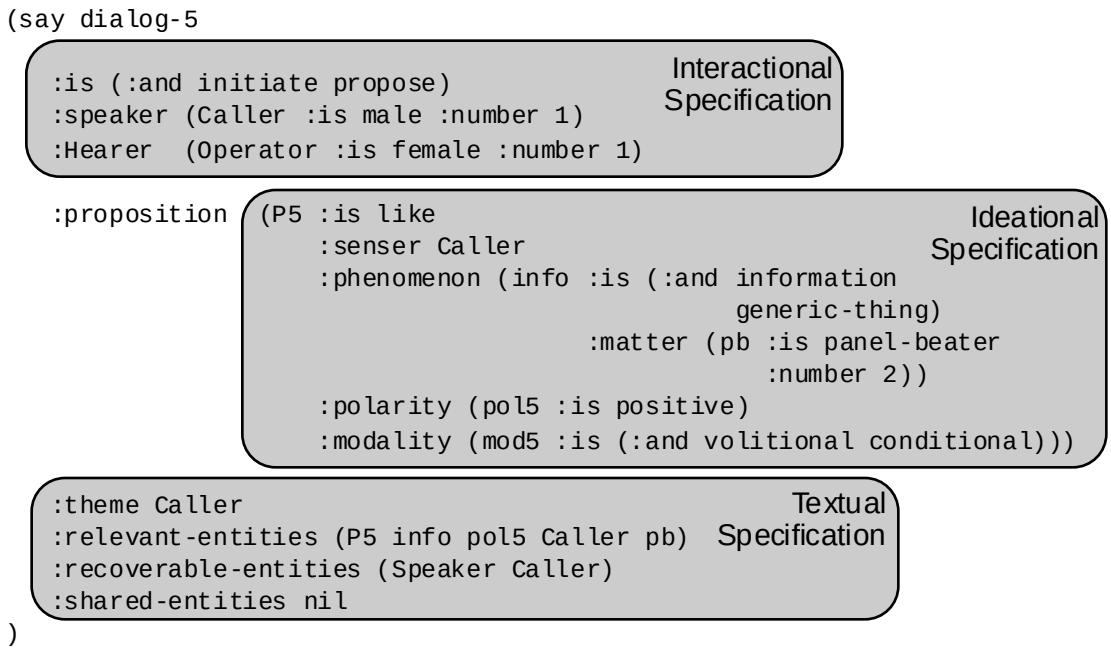


Figure 11.3: The Micro-Semantic Specification for
"I'd like information on some panel beaters."

"say" is the name of the lisp function which evaluates the micro-semantic specification, and calls the generation process.

"dialog-5" is the name of this particular speech-act -- each speech-act is given a unique identifier, its unit-id.

The *:is* field specifies the features of the unit. This is used both for the speech-act as a whole, and for any unit in the ideational content. In this example, the speech-act is provided with a feature-specification *(:and initiate propose)*. The proposition is provided with a single ideational feature: *like*. The feature-specification can be a single feature, or a logical combination of features (using any combination of *:and, :or* or *:not)*. One does not need to specify features which are systemically implied, e.g., specifying *propose* is equivalent to specifying *(:and move speech-act negotiatory propose)*.

## 3.1 Roles of the Speech-Act

Most of the colon-marked fields in figure 11.3 specify the roles of the units, and their filler. For example, the following specifies that the *Speaker* role is filled by an entity with unit-id *Caller*, which is of type *male*.

```
:speaker (Caller :is male)
```

We can specify the filler of a role in two ways:

a) **Unit-Id Only**: We can refer to the unit using just an identifier, e.g., *:Speaker Caller*. If an entity with this name has already been defined, then the Speaker role will point to this entity. If no entity of this name has been defined, then a new entity is defined and inserted into the knowledge-base.

b) **Unit-Definition**: If the role-filler has not been introduced before, we can define the entity within the role-filler slot. For instance, *:speaker (Caller :is male)* defines an instance *Caller*, declares the instance to be of type *male*, and makes the *Speaker* role point to this entity. A unit-definition has the structure:

```
(<unit-id>
  :is      <feature-specification>
  :role1   <unit-specification1>
  :role2   <unit-specification2>
  .....   )
```

Units can also be defined separately from the 'say' form, for instance, by pre-loading a knowledge-base (see section 3.2 below).

The possible roles of the speech-act are:

- **Proposition**: the ideational content of the speech-act -- a unit-specification;
- **Speaker**: the unit-specification of the speaking entity;
- **Hearer**: the unit-specification of the hearing entity;
- **Required**: for eliciting moves, the unit-id of the wh- element. This is a pointer to the element of the ideational content which is being elicited;
- **Elicited**: for proposing moves in response to an elicitation, indicates which element corresponds to the Required element in the elicitation. Fragmentary responses may include just the elicited element.

## 3.2 Ideational Specification

One fundamental difference between WAG's input language, and that of Penman, involves the relation between sentence specifications and the knowledge-base (KB). In both systems, the KB is used to represent the world we are expressing, the entities of interest, the processes they partake in, and the relations between these participants and processes.

In Penman, the ideational component of a sentence plan is not part of the KB, but rather a re-expression of the knowledge in a form closer to language. SPLs are constructed with reference to the KB, but there is no necessary correspondence between the form of the knowledge and the form of the SPL. This is important for Penman, since SPLs are designed to work with a variety of different knowledge-base systems. Penman users supply a function to construct SPLs from the knowledge-base. SPLs may also be constructed by hand, without any knowledge-base being attached to the system at all.

### 3.2.1 Generating Sentences Directly from the Knowledge-Base

The WAG sentence generator, on the other hand, is designed to be used hand-in-hand with its own KRS, so the two are more highly integrated. The standard form of a sentence-specification does not itself contain a specification of ideational-structure, rather it contains a *pointer* into the knowledge-base -- the filler of the *:proposition* role is usually the unit-id of an entity already defined in the knowledge-base.[5] The other fields of the sentence-specification are used to tailor the expression of the indicated knowledge.

To summarise, a Penman-based text-generator needs to build an ideational structure re-representing the content of the KB, which the realisation component then operates on, rather than the KB itself, while a WAG-based text-generator just includes in the sentence-plan a pointer into the KB, and the realisation component then refers directly to the KB to generate a sentence.

To demonstrate WAG's approach, we show below the generation of some sentences in two stages -- firstly, assertion of knowledge into the KB, and then the expression of indicated sections of this KB. The following asserts some knowledge about John and

[5] As we will see below, we can actually supply an ideational specification in the *:proposition* slot, but this should be seen as a short-hand form, allowing assertion of knowledge into the knowledge-base at the same time as specifying a sentence.

Mary, about how Mary left a party because John arrived at the party. *tell* is a lisp macro form used to assert knowledge into the KB.

```
; Participants
(tell John :is male :name "John")
(tell Mary :is female :name "Mary")
(tell Party :is spatial)

;Processes
(tell arrival
      :is motion-termination
      :Actor John
      :Destination Party)

(tell leaving
      :is motion-initiation
      :Actor Mary
      :Origin Party)

;relation
(tell causation
      :is causative-perspective
      :head arrival
      :dependent leaving)
```

Now we are ready to express this knowledge. The following sentence-specification indicates that the speaker is proposing information, and that the head of this information is the *leaving* process. It also indicates which of the entities in the KB are relevant for expression (and are thus included if possible), and which are identifiable in context (and can thus be referred to by name). The generation process, using this specification, produces the sentence: *Mary left because John arrived.*

```
(say gramm-met1
     :is propose
     :proposition leaving
     :relevant-entities (John Mary arrival leaving causation)
     :identifiable-entities (John Mary))

   =>  Mary left because John arrived.
```

As we stated, this approach to sentence specification does not require the sentence-specification to include any ideational-specification, except for a pointer into the KB. The realisation operates directly on the KB, rather than on an embedded ideational specification.

Different sentence-specifications can indicate different expressions of the same information, including more or less detail, changing the speech-act, or changing the textual status of various entities. The expression can also be altered by selecting a different entity as the head of the utterance. For instance, the following sentence-specification uses the *cause* relation as the head, producing a substantially different sentence:

```
(say gramm-met2
     :is propose
     :proposition causation
     :relevant-entities (John Mary arrival leaving causation)
     :identifiable-entities (John Mary))

   =>  John's arrival caused Mary to leave.
```

### 3.2.2 Ideation Specified within Sentence Specifications

Sometimes it is more convenient to specify ideational content within the sentence specification, as in Penman's SPLs. WAG allows this form of expression also: if the filler of the *:proposition* field is an ideational specification rather than a unit-id, then the specification is asserted into the KB, and generation proceeds from there. This approach was exemplified in figure 11.3 above.

## 3.3 Textual Specification

The sentence-specification includes several fields which specify various textual statuses of the entities in the knowledge-base:

### 3.3.1 Theme

This field specifies the unit-id of the ideational entity which is thematic in the sentence. If a participant in a process, it will typically be made Subject of the sentence. If the Theme plays a circumstantial role in the proposition, it is usually realised as a sentence initial adjunct. WAG's treatment of Theme needs to be extended to handle the full range of thematic phenomena.

### 3.3.2 Relevant-Entities

This field contains a list of the ideational entities which are in the *relevance space* (see chapter 5), and are thus selected for expression. In the example in figure 11.3, five entities are nominated as relevant:

```
:relevant-entities (P5 info pol5 Caller pb)
```

This field is not necessary when an explicit ideational specification is included in the 'say' form. In such cases, the generator assumes that all the entities included within the specification are relevant, and no others.

However, when the *:proposition* slot contains only a pointer into the knowledge-base, the *:relevance* field specifies which elements of the KB to express. See chapter 5 for an example using the relevance space to select out successive chunks of a KB (there called a macro-ideational structure).

### 3.3.3 Recoverable-Entities

This field contains a list of the ideational entities which are recoverable from context, whether from the prior text, or from the immediate interactional context (e.g., the speaker and hearer). See chapter 5 for detail.

### 3.3.4 Shared-Entities

This field contains a list of the ideational entities which the speaker wishes to indicate as known by the listener, e.g., by using definite reference. See chapter 5 for details.

## 3.4 Additional Fields of the Input Specification

Some additional fields are allowed in the micro-semantic specification, extending the expressive power of the input language.

### 3.4.1 Constraint

This *:constraint* field allows the user to assert structural information which cannot be expressed by simply specifying features or roles of elements of the speech-act. For instance, tense/aspect is specified in the WAG system by specifying the relative ordering of three points of time (following Reichenbach 1947):

**Speaking-Time**: when the utterance is made;

**Event-Time**: when the event takes place;

**Reference-Time**: A reference point adopted by the speaker.

We can provide a *:constraint* field in the micro-semantic specification to express these relations:

```
(say utterance-1
   :is (:and initiate propose)
   :proposition P1
   :speaker Caller
   :constraint (and (< Proposition.Event-time Reference-time)
                    (= Speaking-time Reference-time)) )
```

### 3.4.2 Preselect

This field allows the user to 'preselect' features of the lexico-grammatical structure. Some grammatical decisions may not be semantically constrained, and this field allows the user to specify which feature to choose, e.g.,

```
:preselect ((p3 indefinite-pronom-group))
```

The first element of a preselection specification (*p3* in this example) is the unit-id of an ideational unit, the second is a feature-specification which the grammatical realisate of the ideational unit must have. This feature-specification must not conflict with the rest of the constraints on that unit, meaning that it must be compatible with the usual lexico-grammatical preselections, and also with the feature's selection-constraint.

### 3.4.3 Prefer

While *:preselect* specifies features that particular grammatical units *must* have, *:prefer* allows the user to specify feature defaults, e.g.,

```
:prefer (passive)
```

During the generation process, there is often more than one feature in a system appropriate to express the semantic specification. This is true when no feature in the system is preselected, and the selection-constraints on more than one feature are met. In these cases, an arbitrary choice needs to be made. By placing a feature in the *:prefer* field, the user can cause the preferred feature to be chosen in such cases.

Feature preferences can also be set globally using the *feature-preferences* variable. See section 4.3.1 below. This variable is also used for semantic defaulting, as discussed just below.

## 3.5 Simplifying the Micro-Semantic Specification

Hovy (1993) points out that as the input specification language gets more powerful, the amount of information required in the input specification gets larger and more

complex. The Penman system uses a couple of methods to avoid the growing complexity of the input specification. These have been adapted to use in WAG as follows.

### 3.5.1 Semantic Defaulting

When the input-specification leaves particular semantic systems unresolved, Penman chooses a feature on a default basis. For instance, the following features are the default when not stated in the input specification: Speech-function: *statement*; Tense: *simple-present*; Polarity: *positive*; Modality: none.

WAG also uses feature defaults. A variable *feature-preferences* is defined, which holds a list of the default (or preferred) features. Before generation begins, the processor goes through each unit of the micro-semantic specification and ensures that, for those systems with no preselected choice, the default feature is selected, if its selection-constraint is met.

Defaulting is necessary since the WAG system uses a deterministic generation strategy -- each grammatical choice must be resolvable as it is met (see section 2.5.1 above). Grammatical choices will not be resolvable unless the semantic decisions they depend on have already been resolved. WAG thus forces those semantic decisions which have not been resolved by the input specification.

Below is shown the Say form from figure 11.3, this time in a reduced form relying on defaults:

```
(say dialog-5
   :speaker Caller
   :proposition
      (P5 :is like
          :senser Caller
          :phenomenon (info :is (:and information generic-thing))
                            :matter (pb :is panel-beater
                                        :number 2))
          :modality (mod5 :is (:and volitional conditional))))
```

### 3.5.2 Macros

Penman allows the user to define *macros* -- short forms in the input specification which expand out to more extensive forms. For instance...

```
:tense present-continuous
```

...in an input specification is replaced with the following before processing begins:

```
:speech-act-id
    (?sa / Speech-act
        :speaking-time-id (?st / time
                   :time-in-relation-to-speaking-time-id ?st
                   :time-in-relation-id (?st ?et ?st) ?et
                   :precede-q (?st ?et) notprecedes))
        :event-time (?et / time
                      :precede-q (?et ?st) notprecedes))
```

WAG does not use macros. To serve the same function, we can add features to the networks which represent complex specifications. For instance, a system could be added to the speech-act network, which could include features such as *present-continuous*, *past-perfect*, etc., each feature being associated with realisations which would assert the necessary structural constraint (see figure 11.4). These features can then be included in the feature-specification of the speech-act, acting as a short-form for the associated structural constraint.

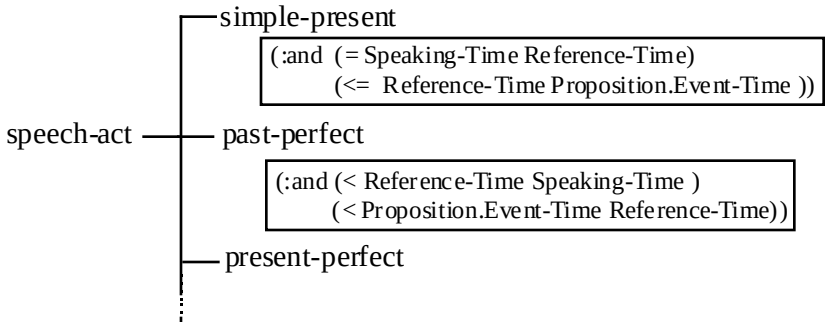


Figure 11.4: Adding Features as a Form of Macro

### 3.6 The Speaker and Hearer Roles

The Speaker and Hearer fields are presently used for two purposes:

- **Pronominalisation**: The Speaker and Hearer roles are used to test if pronominalisation is appropriate: if the fillers of these roles are also part of the proposition being expressed, then pronominalisation is called for, e.g., I, you.
- **Voice Selection**: WAG checks the gender feature of the Speaker to determine which voice to use in Macintosh's text-to-speech system.

Note that the attributes of the Speaker and Hearer do not need to be re-defined for each sentence. We can pre-define the speech-participants as entities in the knowledge-base. Each speech-act specification thence only needs to refer to the unit-id of the speaker and hearer.

In theory, the Speaker and Hearer fields are available for user-modelling purposes: lexico-grammatical choices can be constrained by reference to attributes specified in the Speaker and Hearer roles (cf. Paris 1993; Bateman & Paris 1989b; Hovy 1988a). Since the fillers of the Speaker and Hearer roles are ideational units, they can be extensively specified, including their place of origin, social class, social roles, etc. Relations between the speaker and hearer could also be specified, for instance, parent/child, or doctor/patient relations. Lexico-grammatical decisions can be made by reference to this information: tailoring the language to the speaker's and hearer's descriptions. This has not, however, been done at present: while the implementation is set up to handle this tailoring, the resources have not yet been appropriately constrained.

## 4. Stages in Systemic Sentence Generation

This section describes the algorithms for sentence generation used in the WAG system. These algorithms are fairly identical to Penman's at a gross level, but differ in the way these steps are implemented. A list of the ways the WAG implementation improves on the Penman system are given at the end of the chapter. Note that WAG doesn't include any code from Penman, it is a total re-write.

## 4.1 The General Algorithm

WAG's sentence generation algorithm is shown in figure 11.5. Each of these steps will be discussed below.

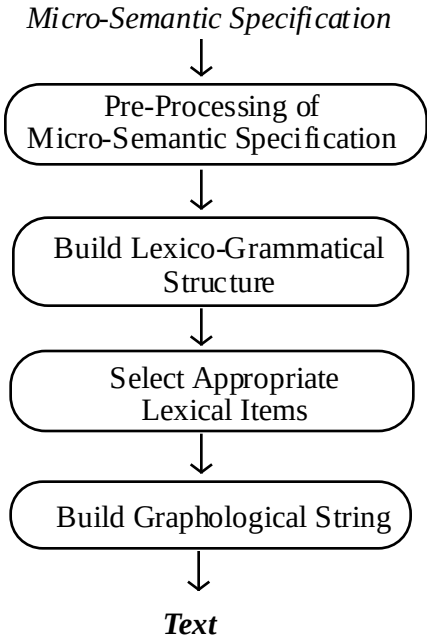


Figure 11.5: The Sentence Generation Algorithm

## 4.2 Initial Processing of the Input

Before generation begins, the input is processed. This processing involves three steps:

1. **Assertion of the Micro-Semantic Specification into the KRS**: the input specification is 'parsed', analysing it in terms of the various roles and fields, and this information is asserted into WAG's knowledge-representation system.
2. **Deriving Implied Structure:** the program derives any additional structural information it can from the partial specification. For instance,
   - Deriving feature information from asserted roles;
   - Deriving additional roles from asserted features.

   These steps are repeated for each element of the micro-semantic specification, in a top-down manner, until all roles are processed.
3. **Defaulting of Unspecified Choices**: After the prior step, there will still be systems which are unresolved. Some of these systems are defaulted. Only systems containing features drawn upon in the interstratal mapping constraints need to be defaulted -- others can be left unspecified. In those systems which are defaulted, features are chosen arbitrarily,[6] except where the user has expressed a preference (see discussion on feature defaulting above).

The result of the input processing stage is what I term a *fully-specified* micro-semantic form. 'Fully-specified' refers to the fact that -- in each unit of the micro-semantic representation -- the features which are relevant for lexico-grammatical processing have been specified. This is required for deterministic generation, as discussed above.

---

[6] If no user-default is specified, the program takes the *last* feature in a system. This is because many systems have a no-realisation alternative, and Systemicists tend to place these features last. A more intelligent program would automatically discover the no-realisation alternative.

## 4.3 Lexico-Grammatical Construction

The goal of the Lexico-Grammatical Construction stage is to build a lexico-grammatical structure which encodes the micro-semantic input. Section 2 above compared two different control strategies for lexico-grammatical construction: *source-driven* and *target-driven*. The WAG system, in common with most Systemic generators, is target-driven -- the construction is based on expanding the lexico-grammatical representation (constrained by the micro-semantics), rather than by realising the micro-semantic representation.

Figure 11.6 shows the basic algorithm behind lexico-grammatical construction in WAG. It defines a top-down, breadth-first, left-to-right construction process (see chapter 9 for a description of these terms). In other words, we first build the structure of the top-most unit (the clause or clause-complex), and then build the structure of each of the unit's constituents, and so on down to word-rank units. This type of generator can thus be called a 'rank-descent' generator.

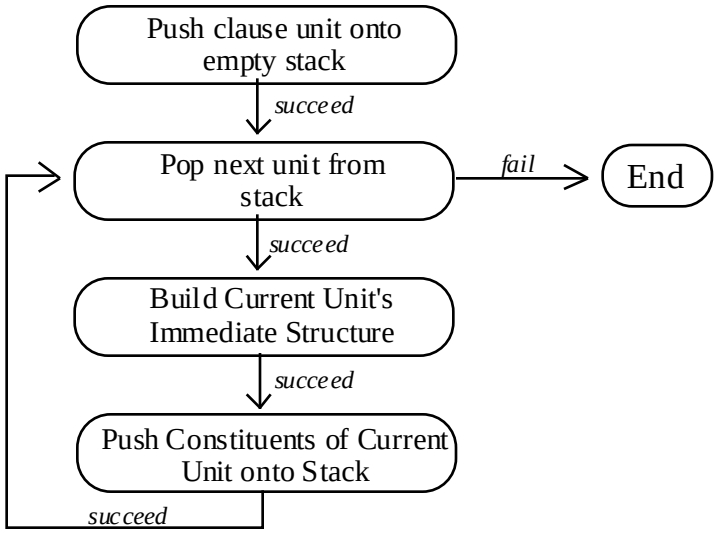


Figure 11.6: WAG's Lexico-Grammatical Construction Process

This algorithm uses a *stack* data-structure. A stack is a data-structure used for storing items. It is basically a last-in, first-out queue. You 'push' an item onto the stack -- place an item at the front of the queue. You can push other items on top of this. You can also 'pop' an item, meaning that you take the item from the top of the stack. See figure 11.7.

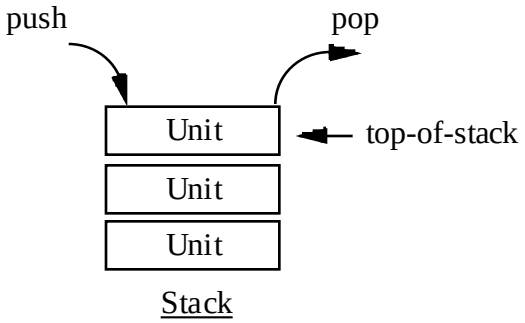


Figure 11.7: The Unit Stack

The stack is used to store the constituents-to-be-processed. The process starts off with only one element on the stack -- the sentence unit. At this point, the information in this

unit is minimal, just a specification that the unit is a clause[7], and a pointer to the *Referent* (semantic content) that this clause-unit is expressing.

Processing then begins: the top element is 'popped' off the stack, the system network is traversed to build up its feature-list, and the realisation statements associated with these features are applied, thus building the immediate structure of the unit.

When the element's immediate structure is complete, we then need to complete the structure of each of its constituents. So we push each of these constituents onto the Unit-Stack, and cycle back to the beginning of the process: pop the next unit off the stack, process this, and so on.

We continue popping and processing units until there are no units left to process. This occurs when all constituents of the sentence-tree have been fully specified. We thus go on to the bubble labelled 'End' in figure 11.6. We are now ready to move onto the next stage of the generation process -- lexical selection[8].

### 4.3.1 Immediate-Structure Construction

I will now provide more detail about the immediate-structure building stage of the generation process. Following sections will focus on two aspects of this stage -- forward-traversal and constituency ordering.

The construction within each unit of the target is resource-driven -- controlled by the traversal through the system network (from left to right). In each system, the program chooses a feature whose semantic constraints are compatible with the micro-semantic input. The structural realisations of this feature are then asserted, and the process advances to the next enterable system. When all enterable systems are processed at that rank, the unit is complete. Figure 11.8 shows the algorithm for generating the immediate structure of a unit. It is reasonably similar to the flowchart proposed by Matthiessen & Bateman (1991, p106), but has been developed separately.

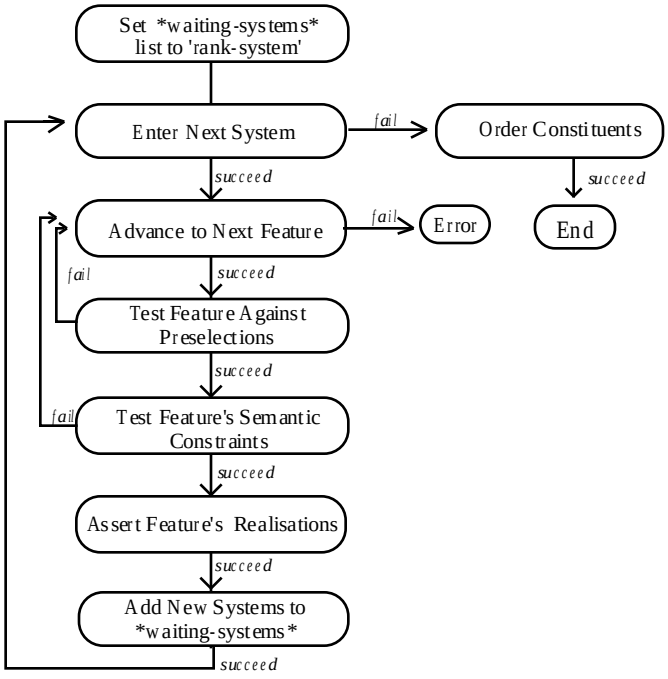


Figure 11.8: The Immediate-Structure Building Algorithm

[7] The feature *clause* leads on to both *clause-simplex* and *clause-complex.*

A brief summary of each of these steps follows:

1. **Set *Waiting-systems* list to 'rank-system'**: the variable *Waiting-systems* contains the list of systems which are waiting for processing, i.e., those systems whose entry conditions are satisfied at the present point of traversal, but which have not yet been 'entered' (no feature has been selected as yet).
2. **Enter next system**: The next system from the *Waiting-systems* list is retrieved, becoming the *current-system*. At this point, the features of the system are ordered by various means.
   a) **Initial ordering**: Internally, the features of a system are ordered as they appear in the system definition. The user can set a variable *feature-selection-mode* which will change this default ordering in two ways:
      • **Reverse-order:** the list of features is reversed before further processing. This is useful for sentence generation, since the last feature in a system is usually the one with no realisation attached.
      • **Random-order**: the list of features is jumbled.
   b) **Preferential ordering**: The user can specify a list of 'preferred features' -- features that will be considered before any other features. These preferred features are put at the front of the list resulting from (a) above.
3. **Advance to next feature**: the next feature from the system is selected. If there are no more features in the system (no valid alternatives), then processing fails -- either the resources contain inconsistencies, or we have reached a generation gap.
4. **Test feature against preselections**: the feature is tested against the preselections for this unit. If the feature is consistent with the preselections, processing continues, else the process returns to (3) to try another feature from the system.
5. **Test feature's semantic constraints**: the feature's selection constraint (see chapter 6 on inter-stratal mapping) is tested, and if it is consistent, it is chosen, otherwise the process goes back to try another feature from the system.
6. **Assert feature realisations**: the realisations of the feature are asserted, with the exception for the *:order* and *:partition* rules, which are stored for later application. These realisations are applied at the end of the traversal, when we know which roles conflate, which are presumed, and which of the optional roles were actually inserted. If the assertion of the realisations fails (the realisations are inconsistent with grammatical information from other features), an error is signalled. This should not happen as Systemic grammars should be so constructed in a way that any legal combination of features is realisable.
7. **Add new systems:** The program checks for any systems which become enterable with the addition of the chosen feature. These systems are added to the *Waiting-systems* list. Penman and WAG both keep a list, for each feature, of the entry-conditions which the feature appears in. Thus, after selecting a feature, we do not need to check all systems to see if they have become enterable, but only those on this list. Also, any systems which have already been entered are automatically ignored.
8. **Order constituents**: When all systems have been processed, the sequence rules (order and partition) are processed, placing the units in their surface ordering. See section 4.3.3 below for more details.

---

[8]The lexical selection process could be performed intermixed with the lexico-grammatical construction. If so, then the processor would then advance to graphological realisation.

### 4.3.2 Forward-Traversal Algorithms

I have outlined Systemic generation as forward-traversal through the system network. If there were no simultaneous systems in a system network, traversal would be a simple matter of selecting a series of features in a single path from root to leaf. However, networks allow simultaneous systems, so the order in which systems are processed is not totally determined. Simultaneous systems can be entered in any arbitrary order. This gives rise to two alternative strategies for network traversal:

1) **Depth-first**: The systems which extend from the last selected feature are processed before simultaneous systems. Traversal follows one branch of the network to the leaves before exploring others. Depth-first traversal is achieved by placing newly activated systems at the *front* of the *waiting-systems* list, so that they will be processed first.
2) **Breadth-first**: Systems at the same systemic depth are processed before the systems which depend on them. Breadth-first traversal is achieved by placing newly activated systems at the *end* of the *waiting-systems* list, so that they will be processed last. Systems which were already on the list represent simultaneous systems, and they will be processed first.

The choice between these strategies doesn't affect processing efficiency, since all entered systems have to be processed anyway. The order of entry should not affect the results of the generation process.

### 4.3.3 Sequencing of Constituents

This section describes the algorithm used to sequence grammatical constituents in the WAG generator. It represents a very succinct method for sequencing units systemically. Sequencing is applied after the traversal is complete, since it is only at this point that we can be sure which of the elements marked as optional are actually included, and also which functions conflate together.

The WAG formalism uses two sequence operators:

- **order**: indicates absolute ordering (adjacency), e.g., *(:order A B C)* indicates function A immediately precedes function B, which immediately precedes function C.
- **partition**: indicates relative ordering, e.g., *(:partition A B)* indicates function A precedes function B, but not necessarily adjacently.

**Optionality**: Elements of a sequence rule can be *optional* (need not actually occur in the final structure). Optional elements are designated by being parenthesised, e.g., *(:order Subject Finite (Negator) )*.

**Front & End**: The sequencing of a unit in relation to the front and end of the grammatical unit can be indicated by inclusion of pseudo-functions 'Front' and 'End' in the sequence rule, e.g., *(:order Front Subject ), (:order Punctuation End).*

To exemplify the processing, I will assume a clause which, after all systems are entered, has the following sequence rules:

```
Order:      Punct ^ End;
            Pred ^ Object;
            Subject ^ Finite ^ (Negator)
Partition:  (Modal) # (Perf) # (Prog) # (Pass) # Pred
```

**1. Sequence Rule Preparation**: The order and partition rules are standardised through three steps:

a) **Removal of optional elements**: the order/partition rules may contain optional elements. Any optional element which is not present in the structure is removed from the rule. The sequence rules shown above simplify to those below:

```
Order:      Punct ^ End;
            Pred ^ Object;
            Subject ^ Finite
Partition:  Modal # Pred
```

b) **Standardisation of Role-Labels**: Each constituent may have multiple role labels (due to conflation). Different rules may refer to one constituent using different role labels, e.g., assuming that *Finite* and *Modal* are conflated, then the final two sequence rules in the set from above contain references to one unit using different role-names. The order/partition rules are standardised so that only one role per role-bundle is used.

```
Order:      Punct ^ End;
            Pred ^ Object;
            Subject ^ Finite
Partition:  Finite  #  Pred
```

c) **Splitting into two-element rules**: Each sequencing of more than two elements is split into a number of binary sequencers. The rules of this example are all binary after the elimination of the optional elements, but this is not always the case. For example:

```
Finite # Prog # Pred  =>    Finite # Prog;  Prog # Pred
```

**2. Processing of Sequence Rules**

To merge the information contained in these sequence-rules, an *ordering graph* is used -- a data-structure which represents the ordering between any pair of elements. The sequencing rules are applied to the graph one at a time, as shown in the following example.

a) **Initial State**: The units start out unordered in respect to each other, but ordered in respect to the front (FRONT) and end (END) of the unit, as demonstrated in figure 11.9(a). In these ordering graphs, a continuous line between roles indicates adjacency, a line broken by a || indicates that other elements may intercede (partitioned).

b) **Order Cycle**: Each order rule is applied in turn. The successive effect on the order graph is shown in Figure 11.9(b-d).

c) **Partition Cycle**: Each partition rule is then applied. Figure 11.9(e) shows the application of the one partition in this example.

d) **Reading off the Sequencing**: After all the sequence rules are applied, we can read off the sequencing from the graph. In most cases, there is only one path through the graph. However, in some situations, sequence is not totally determined, and alternative orderings may be possible (for instance, the WAG grammar does not totally determine the sequence of multiple Circumstances, or nominal Qualifiers). In such cases, the process just takes the first ordering alternative.

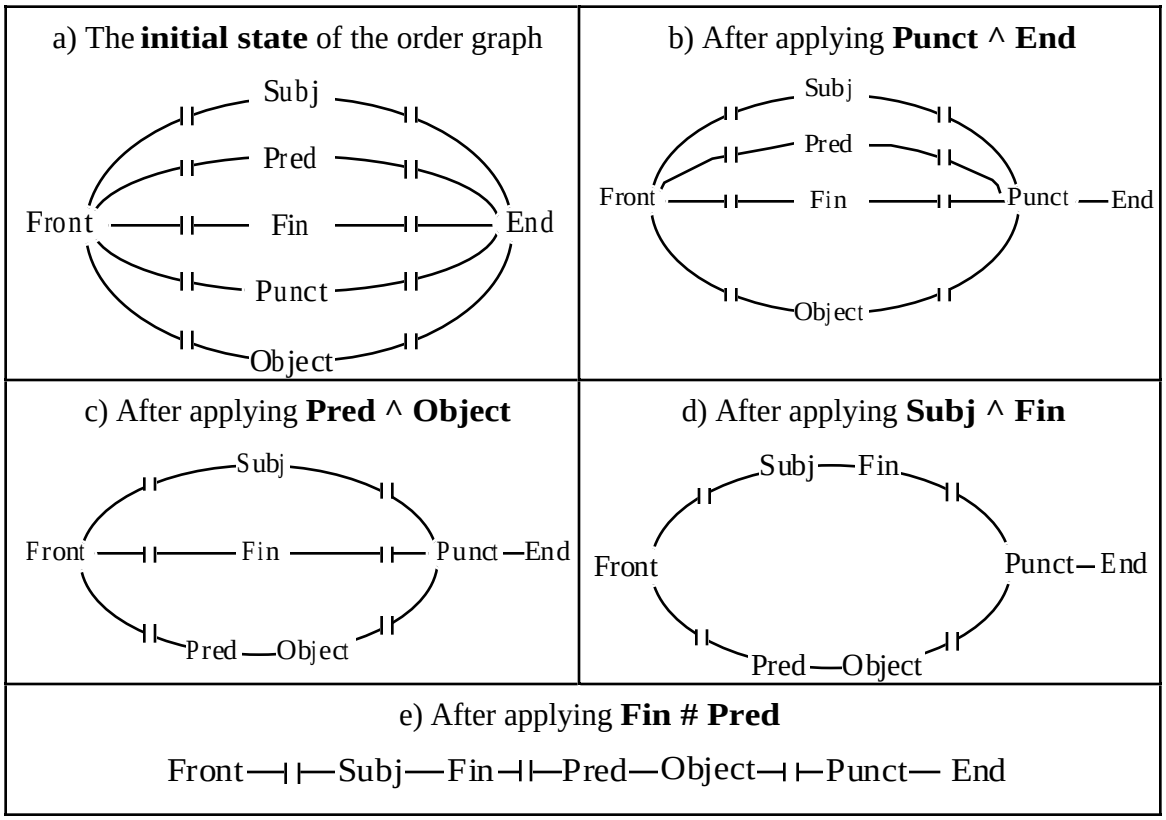


Figure 11.9: Successive States of the Order Graph

## 4.4 Lexical Selection

In Penman's lexical selection algorithm, semantic filtering is applied first: the semantics provides the set of candidate lexemes which express the ideational type of the referent. These candidates are then filtered grammatically, and one of the remaining candidates is chosen:

> "Abstractly, there are two ways in which sets of candidate lexical items are constrained and denotational appropriateness is the first kind of constraint applied. Then grammatical constraints -- such as the requirement that the lexical item be an en-participle -- are used to filter the set of denotationally appropriate terms." (Matthiessen 1985).

The WAG system filters on grammatical grounds first. As a general case, I believe that the Penman approach (semantic filtering first) is best. However, for a variety of reasons, lexical selection in WAG is quickest when grammatical filtering is performed first. This is true particularly for closed-class lexical-items (pronouns, prepositions, conjunctives, verbal-auxiliaries, etc.), of which most can be totally resolved through grammatical selection only. Even for open class items, grammatical filtering seems to be quicker.

The Penman system associates each lexeme with a single ideational feature (or *concept* in Penman's terms). The WAG system overcomes this limitation, allowing each lexeme to be associated with a *set* of ideational features. For instance, to specify the semantics for the word "woman", Penman would need to create an ideational feature *woman*, which inherits from both *female* and *adult*.[9] In the WAG system, we can specify that the

[9] It is not necessary to specify the feature *human*, since *female* in the Penman Upper Model inherits from *human*.

lexeme's semantics is *(:and female adult)*, avoiding the need to create a new concept for each combination of features.

## 4.5 Text and Speech Output

The sentence generator can produce either text (a graphologically formatted sentence), or speech output.

1. **Text Output**: The text output is derived from the lexico-grammatical structure (including lexical items) constructed during the prior stages. The mappings between lexico-grammatical and graphological form are not stated declaratively -- they are encoded in the lexico-grammar-to-text procedure. These resources will eventually be declarativised. The graphological string is derived as follows:
   a) the lexical items are extracted from the leaves of the lexico-grammatical tree, in order of occurrence from left to right.
   b) an appropriate graphological form is generated for each lexeme, given its inflection feature.
   c) the left-most graphological form is capitalised.
   d) Spacing: a space character is placed between each graphological form. Some punctuation symbols modify this rule:

      No space before: **. , ; ? ! ' "** (close quotes)

      No Space after: **' "** (open quotes)
2. **Speech Output**: If the speech-output option is selected, the WAG system speaks the generated sentence using the Macintosh Speech Manager (a text-to-speech program). WAG will check the designated gender of the 'Speaker' role of the input specification, and choose a voice appropriately.

# 5. Comparison With Penman

In building the generation component of WAG, I have borrowed strongly from Penman's general architecture. However, I have attempted to correct many of the short-comings in the Penman system. Note that WAG uses none of Penman's code.

## 5.1 Similarities to Penman

The areas in which WAG has borrowed from Penman are:

1) **Grammar-driven control:** WAG uses Penman's grammar-driven control strategy, in common with the majority of Systemic generators.

2) **Traversal Algorithm:** The WAG traversal algorithm is similar to Penman's, although the choosing of features in a system is different in WAG - based on evaluating feature selection conditions, rather than traversing a chooser-tree.

2) **Upper Modelling**: Penman's Upper Model is the basis of WAG's ideational representation, although WAG's Upper Model is represented as a system network, rather than as a LOOM inheritance network. WAG has also adopted Penman's means of handling domain knowledge - subsuming domain concepts under upper-model concepts rather than relating them directly to the lexico-grammar (see chapter 3).

3) **Resource Definition and Access**: For reasons of resource-model compatibility between Penman and Nigel, WAG accepts system definitions in Penman's format, and can export to this format (although a modified format is

preferred for WAG). Many of WAG's resource-access functions (functions for accessing the stored Systemic resources) are named identically to Penman's, although the internal storage of the resources is different. This has been done for code compatibility reasons.

## 5.2 Differences from Penman

WAG improves on Penman in several directions:

1. **Input Specification**: WAG's micro-semantic specification form is not dissimilar to Penman's input form -- Sentence Plan Language (SPL) -- except for several improvements, which were discussed in chapters 4 and 5. In summary, these are:

   a) **Extended and linguistically-based speech-act network:** the speech-act network has been extended to handle a wider range of speech-acts. The speech-act categories rest on a firm theoretical basis in the Berry-Martin tradition.

   b) **Treatment of proposition as part of speech-act**: The relation of propositional content to speech-act was rather ad-hoc in Penman, where the speech-act was tacked on to the proposition to be expressed. Speech-function seems to have been added on as an afterthought to an originally declarative-only system. In the WAG system, the relationship has been clarified, with the propositional content being treated as a role of the speech-act to be expressed.

   c) **Generation directly from KB**: WAG allows sentence-specifications to include just a pointer into the KB, while Penman requires an ideational structure to be specified within each sentence-specification.

   d) **Designation of Wh-element in elicitations**: it is difficult to designate the element which should be the wh-element of a question in Penman, the user needs to directly specify the answer to an identification inquiry, a process which involves some knowledge of the internal working of the Penman system. WAG allows the user to designate the wh-element (the Required element) in the input specification, in a simple, theoretically-based manner.

   e) **Extended textual specification**: WAG has extended the range of textual specification possible in the input form. This includes the recoverability, identifiability, and relevance of entities. To match these features in SPL, the user needs to include inquiry preselections -- forced responses to Penman's inquiries -- a rather low-level approach.

   f) **Complex ideational feature specifications:** In Penman, each ideational unit can have only a single feature (e.g., *ship* ), or at most a conjunction of features. WAG allows the user to specify the type of semantic unit using any logical combination of features, using conjunction, disjunction or negation.

2. **Interstratal Mapping:** Penman's chooser-inquiry interface has proven problematic for two reasons:

   a) The chooser-inquiry interface is partially procedural, and thus not re-usable for analysis. The WAG system has replaced the chooser-inquiry interface with a declarative mapping system (following Kasper's approach), used for both analysis and generation.

   b) When extending Penman's resources to generate new sentences, it is often difficult to work out what input specification is necessary to get a particular lexico-grammatical form. The main reason for this is the need to work on four levels of representation:

      • Upper-model concepts and structures

      • Inquiry specifications

      • Chooser Trees

      • The System Network

      The WAG system simplifies the mapping process by mapping directly from grammatical features to upper-model concepts. It is thus easier to discover what input specification is needed to produce a particular lexico-grammatical structure.

3. **Structure Building**: WAG improves on Penman's structure building in several ways:

   a) **Conciseness**: The Penman code for lexico-grammatical construction is quite long and involved, having been developed by several programmers over ten years. Some parts are difficult to penetrate, even by those who maintain it. The WAG system has the advantage of being designed rather than evolved, and takes advantage of the progress made in Penman. It has also been implemented by a single programmer, so is more highly integrated.

   b) **Sequencing**: The Penman formalism system uses four sequencing operators (order, partition, order-at-front, order-at-end). However, most ordering is actually derived from a set of default ordering rules. These default orderings are not part of the Systemic formalism, but rather an ad-hoc extension. Penman's sequencing information is not sufficient for parsing, since the resources provide mostly default ordering, rather than all possible orderings.

      The WAG system has extended the sequencing formalism to allow optional elements in sequence rules. All ordering in the WAG grammar is done without the default ordering resource. The WAG grammar is thus suitable for parsing as well as generation.

   c) **Proper handling of disjunctive preselections**: Penman does not handle preselections properly where the preselection includes some disjunction. The WAG system corrects this problem.

   d) **Use of a generalised KRS**: The Penman system has specialised code for dealing with realisation rules. All of WAG's processing is based on top of WAG's KRS, which is used for asserting realisation statements during generation or parsing, for asserting or testing feature selection conditions, and for asserting knowledge into the knowledge-base.

4. **Lexical Selection**: The Penman system associates each lexeme with a single ideational concept. The WAG system overcomes this limitation, allowing each lexeme to be associated with a set of ideational features.

5. **Lexical network into system network**: In Halliday's Systemic grammar, lexical features are organised under the lexico-grammar system network - the

word-rank sub-network. Penman does not follow this approach.[10] For generation purposes, the lexical features are not organised in terms of a network, and Penman cannot check on the inheritance relations between lexical features. The features are organised into an inheritance network only for lexical acquisition (see Penman Project 1989), and this information is organised in a Loom inheritance network, rather than as part of the Nigel lexico-grammatical network. The WAG system incorporates the lexical features into the lexico-grammatical network, under the *word* feature. This resource is used in processing to test compatibility of lexical features.

6. **Single formalism for all levels**: The WAG system uses the same knowledge representation system for all structural representation, including the internal representation of the micro-semantic input (speech-act and ideational content) and lexico-grammatical form. Penman has two systems - Loom is used to represent knowledge, and Penman provides its own internal knowledge representation system for representing lexico-grammatical structures.

# 6. Conclusions

While the WAG generator has only been under development for a few years, and by a single author, in many aspects it meets, and in some ways surpasses, the functionality and power of the Penman system, as discussed above. It is also easier to use, having been designed to be part of a *Linguist's Workbench* -- a tool aimed at linguists without programming skills.

The main advantage of the Penman system over the WAG system is the extensive linguistic resources available. While the WAG system can work with the grammar and lexicons of the Nigel resources, I have not yet connected these resources to the micro-semantics, so micro-semantic generation using Nigel is not yet possible. The writing of appropriate feature selection-constraints is a task for future development.

This chapter has also provided extended discussion on the internal workings of a Systemic generator, for which there is scant literature (although see Patten 1985; Matthiessen & Bateman 1991).

[10] John Bateman has a version of Penman which partially corrects this problem.