Interpreting Systemic Grammar as a Computational Representation: A Problem Solving Approach to Text Generation

Terry Patten

Ph.D.

University of Edinburgh

1986

TO MY PARENTS

Acknowledgements

I would like to express my sincere gratitude for the constructive criticism and guidance patiently provided by my thesis supervisor Graeme Ritchie throughout the development of this work. I would also like to thank my other supervisor Austin Tate, and the rest of the Planning Group, for providing insights into AI problem solving. Thanks must also go to Jim Howe and Henry Thompson for helping me to get underway on this project; to Mark Drummond, Andy Golding, and Chris Sothcott for valuable technical discussions; and to Mark Kingwell for proof-reading a draft of this thesis. Of course, I am solely responsible for any mistakes and ommissions herein.

This work was supported in part by Alberta and Canada Student Loans, and by an Overseas Research Student award.

Declaration

I hereby declare that this thesis has been composed by myself, and that the work reported in it is my own.

Terry Patten

Abstract

A new approach to automatic natural-language text generation is described. The approach exploits artificial intelligence (AI) problem-solving techniques and explicitly represents the grammars using an established linguistic formalism. It is demonstrated that AI problem solving and Halliday's theory of systemic grammar share some fundamental properties, and that this has resulted in an equivalence between the representations found in these two fields. The equivalent representations mean that a systemic grammar can be translated trivially and automatically into AI knowledge-representation languages and then used as a linguistic knowledge base. This knowledge base can be put at the disposal of a powerful, general-purpose problem solver which applies goal-directed knowledge-based techniques to selectively and efficiently generate text. A significant linguistic characteristic of this approach is that it allows the incorporation of the socio-semantic level of systemic theory exactly as described in Halliday's recent writings. This is also of major computational significance because the socio-semantic level acts as highly-compiled knowledge that guides the grammatical problem solving. The approach thus exploits the state-of-the-art computational techniques while manifesting an established linguistic theory.

The approach is ideal for generating explanations in expert systems because the same problem solver that applies the expert knowledge to problems can also apply the linguistic knowledge in the grammar to text-generation problems. This not only supplies a powerful and efficient mechanism for the text-generation problem, but also greatly simplifies the system as a whole.

Although any one of several AI knowledge-representation languages could have been used to represent systemic grammars, production rules were chosen for this project. Since systemic grammars can be trivially translated into production rule form, a formalization of systemic grammars can be given in terms of productions and derivations, resembling the traditional formalization of structure grammars. This formalization of systemic grammar is part of the formal model of the text-generation method which, together with a sample implementation, serves to clarify and illustrate the approach.

CONTENTS

Chapter 1. Introduction.....1
1.1. General overview.....1
1.2. Scientific context.....1
1.2.1. Major context: AI Text Generation.....2
1.2.2. Minor context: functional linguistics.....4
1.3. Important assumptions.....4
1.4. Specific overview.....4

Chapter 2. Background I: AI Problem Solving.....7
2.1. Search.....7
2.1.1. Brute-force search.....8
2.1.2. Heuristic search.....8
2.1.3. Forward chaining.....9
2.1.4. Goal-directed backward chaining.....10
2.2. Compiled problem-solving knowledge.....11
2.2.1. Sources of compiled knowledge.....11
2.2.1.1. Precompiled knowledge and compilation by hand.....12
2.2.1.2. Automatic knowledge compilation.....12
2.2.2. Reasoning from first principles.....13
2.3. Summary.....13

Chapter 3. Background II: Systemic Grammar.....15
3.1. History.....15
3.1.1. Malinowski.....15
3.1.2. Firth.....16
3.1.3. Halliday.....17
3.2. The goals of systemic grammar.....17
3.3. Important concepts in systemic grammar.....18
3.3.1. Feature.....19
3.3.2. System.....19
3.3.3. System network.....20
3.3.4. Delicacy.....22
3.3.5. Functional analysis.....23
3.3.5.1. Transitivity.....24
3.3.5.2. Ergativity.....25
3.3.5.3. Mood.....27
3.3.5.4. Theme.....27
3.3.5.5. Information.....28
3.3.5.6. Functional analysis for groups.....28
3.3.6. Rank.....29
3.3.7. Realization rules.....31
3.3.7.1. Conflation.....31
3.3.7.2. Expansion.....32
3.3.7.3. Adjacency.....32
3.3.7.4. Lexify.....34
3.3.7.5. Preselection.....34
3.3.7.6. NO insertion/inclusion.....37
3.3.8. The metafunctions.....37
3.3.8.1. The ideational metafunction.....38
3.3.8.2. The interpersonal metafunction.....38
3.3.8.3. The textual metafunction.....38

3.3.8.4. Metafunction and linguistic description.....38
3.3.9. Recursive systems.....39
3.4. The strata.....42
3.4.1. The semantic stratum.....42
3.4.2. The grammatical stratum.....43
3.4.3. The phonological/orthographic stratum.....43
3.4.4. Interstratal preselection.....43
3.5. The semantic stratum.....44
3.5.1. Field.....46
3.5.2. Tenor.....46
3.5.3. Mode.....46
3.5.4. Register and metafunction.....46
3.5.5. A closer look.....48
3.6. Example.....49
3.7. Summary.....51

Chapter 4. The Conflation.....53
4.1. The fundamental relationship.....53
4.1.1. Alternatives in AI problem solving.....53
4.1.2. Alternatives in systemic linguistics.....54
4.1.3. The fountainhead.....56
4.2. The conflation.....56
4.2.1. Conflating representations.....57
4.2.1.1. Conflating gates and forward-chaining rules.....57
4.2.1.2. Conflating systems and backward-chaining rules....58
4.2.1.3. Conflating the grammar and the knowledge base.....60
4.2.2. Conflating text generation with problem-solving.....60
4.2.3. Conflating the semantic stratum with compiled knowledge.....63
4.2.4. Conflating behaviour potential and general problem-solving knowledge.....67
4.3. An example.....68
4.4. Advantages.....79

Chapter 5. Theoretical Issues.....81
5.1. The interfaces.....81
5.1.1. The planner/generator boundary.....81
5.1.2. Separation of semantic/pragmatic and grammatical knowledge.....83
5.2. The functional approach.....84
5.2.1. What is a functional approach?.....84
5.2.2. Formal and functional approaches.....85
5.2.3. A functional computational approach.....88
5.2.4. Explanation and compiled knowledge.....91
5.3. Contrasts with the generative paradigm.....94
5.3.1. Chomsky's modularity hypothesis.....94
5.3.2. The power of the grammar.....95
5.4. Summary.....96

Chapter 6. The Formal Model.....98
6.1. A formalization of systemic grammar.....99
6.1.1. Structure.....100
6.1.2. The language generated by a grammar.....104
6.1.3. Derivation.....106

6.2. The completeness proof....108
6.3. Problem reduction....111
6.3.1. AND/OR graphs....112
6.3.2. System networks and AND/OR graphs....113
6.4. Algorithms....118
6.5. An example....123

Chapter 7. SLANG-I -- The Implementation....126
7.1. Overview....126
7.1.1. The abstract architecture....127
7.1.2. The grammar productions....127
7.1.3. The syntactic structures....130
7.1.4. The control strategy....132
7.1.5. Overview conclusion....133
7.2. SNORT (System Network - OPS5 Rule Translator)....133
7.2.1. The system network notation....133
7.2.2. The production rule notation....138
7.2.3. The translation....143
7.3. SLANG-I....145
7.3.1. Realization productions....145
7.3.2. The support system....150
7.4. Limitations of the current implementation....154
7.5. Alternative implementations....155
7.5.1. Other production systems....155
7.5.2. Inheritance hierarchies....156
7.6. Summary....157

Chapter 8. Comparison with Other Work....158
8.1. The grammar-driven approach....158
8.1.1. PROTEUS....159
8.1.2. Nigel....161
8.1.3. Advantages of grammar-driven systems....162
8.2. The goal-driven approach....162
8.2.1. KAMP....163
8.2.2. MUMBLE....165
8.2.3. Advantages of the goal-driven approach....166
8.3. Combining the approaches....167
8.3.1. TELEGRAM....167
8.3.2. SLANG....169
8.3.2.1. SLANG as a grammar-driven method....169
8.3.2.2. SLANG as a goal-driven system....170
8.4. Problem-reduction in Nigel and SLANG....171
8.4. Summary....173

Chapter 9. Conclusions....174
9.1. Summary....174
9.1.1. The problem....174
9.1.2. The solution....175
9.1.3. Theoretical issues....176
9.1.4. The formal model....177
9.1.5. The implementation....178
9.1.6. Other work....179
9.2. Major problems....179
9.3. Future research....181

9.3.1. Incorporation of SLANG into an expert system.....182
9.3.2. Supplementary linguistic treatment.....182
9.3.2.1. Systemic speech generation.....182
9.3.2.2. Supply missing grammatical ranks.....182
9.3.2.3. Linguistic defaults.....183
9.3.3. Further compilation of the semantics.....183
9.3.4. Reasoning with knowledge at the grammatical stratum..184
9.4.5. Natural-language understanding.....184
9.4. Conclusion.....187

Appendix A. OPS5 Tutorial.....189

Appendix B. Sample Texts.....194
1. Explanation for a hypothetical expert system.....194
2. Sample explanation of a plan.....198
3. Examples from the semantic stratum.....211

Appendix C. The Grammar.....214
1. The clause network.....214
2. The nominal-group network.....231
3. The determiner network.....234
4. The quantifier network.....236
5. The prepositional-phrase network.....238
6. The verb network.....238
7. The noun network.....249
8. The conjunction network.....256
9. The modal adjunct network.....261
10. The adjective network.....264
11. The adverb network.....265
12. The preposition network.....265
13. The semantic stratum.....266

Appendix D. Program Listing.....272
1. The initialization file.....272
2. Realization productions.....272
3. The support system.....275
4. The external LISP operators.....278
5. SNORT.....281

Bibliography.....289

1. Introduction

1.1. General overview

This thesis explores a new approach to text generation that interprets systemic grammar as a computational representation. Systemic grammars are interpreted as domain-specific knowledge and used by an artificial intelligence problem solver to solve text-generation problems. This is made possible by a fundamental, but hitherto unrecognized, relationship between systemic grammar and problem solving. This approach solves the methodological problem of interfacing specialized knowledge-based computational representations with equally specialized linguistic representations--because in this case the representation is the same. Previously, text-generation systems have had to make either linguistic sacrifices for computational reasons or computational sacrifices for linguistic reasons.

This approach to text generation has been investigated through two different channels. The primary means of investigation has been a substantial implementation involving a relatively large systemic grammar. The secondary means of investigation has been the construction of a formal model. Aside from a detailed discussion of the approach to text generation, the implementation, and the formal model, the topics covered in this thesis include the relevant background in AI problem solving and systemic grammar, a discussion of the theoretical issues raised by this approach, a comparison with other work in the field, and a sampling of ideas for future research.

1.2. The scientific context

Work in the area of natural-language processing has appeared under several banners, each of which has associated objectives and assumptions. It is therefore important to clarify the objectives and assumptions of the present work. Perhaps it would be best to begin by explicitly stating some of the fields of study to which no

contribution has been intended.

Some of the work in natural-language processing, and in particular text generation, is intended to have psychological implications (e.g. McDonald, 1980). No such implications are intended here. It is hoped that, like any other artificial intelligence (AI) research, this thesis may provide useful suggestions and concepts for future psychological description (see Ritchie, 1980, p.19).

Some other work in natural-language processing is intended to introduce or develop a linguistic theory (e.g. ibid.). Although the theory of systemic grammar is central to this thesis, no attempt has been made, with the exception of some formalization, to contribute to the existing theory. This point must be emphasized since one of the most important claims that is made here is that an established linguistic theory has been used and has not been tampered with in any way.

Finally, although the state-of-the-art AI problem-solving techniques play a central role in this thesis, no attempt has been made to advance this state-of-the-art. This too must be emphasized since the credibility of this approach to text generation depends on the use of indubitable problem-solving techniques.

1.2.1. Major context: AI text generation

The primary scientific context for this work is the AI field of text generation. Text generation is a subfield of natural-language production although its boundaries are not easy to define exactly. Certainly the bottom end of text generation is the actual text itself, but at the top end the picture is not so clear. It will be assumed that natural-language production consists roughly of two stages that perhaps operate in parallel: text planning and text generation. The text planner is responsible for dividing up and ordering the conceptual input to the language production facility. The text generator takes the resulting chunks of semantic/pragmatic representation and transforms them into the desired natural-language (English will be assumed throughout the thesis).

There are two major text-generation objectives that will be stressed here. The first is that the text generator should include an explicit grammar written in an established linguistic formalism. This allows the grammar to be written, understood, modified, judged and so on, independently of the rest of the text-generation system. It also facilitates linguistic contributions to the project from other sources.

A second objective of AI text generation is to develop systems that are practical. The current interest in expert systems, and the important claim that expert systems can explain their reasoning, means that proficient text-generation systems are urgently needed for practical application. The urgency is increasing as expert systems are adopted in socially oriented domains such as medicine and law. To be practical in this sense, the text generator must have good linguistic coverage, and it must be fast. Given that the linguistic coverage will be provided by a linguistic grammar (see previous point), the coverage is really a linguistic problem and will not be addressed here. The speed of the generation is, however, extremely relevant. This constraint requires that the text generation be controlled by the most sophisticated computational techniques available. For this kind of problem, the most sophisticated and efficient computational techniques are those used in AI knowledge-based problem solving.

Unfortunately, there appears to be a conflict in the two objectives just mentioned. Established linguistic formalisms use highly specialized representations developed for linguistic purposes. AI problem solving also uses highly specialized representations, but these were developed for computational purposes. The problem is that these two sets of highly specialized representations (not surprisingly) appear to be incompatible. A crucial objective of AI text-generation research is thus to interface the linguistic and problem-solving representations so that the other two objectives can both be met.

1.2.2. Minor context: functional linguistics

The nature of the text-generation task--finding grammatical constructs that satisfy semantic/pragmatic goals--suggests that a functional linguistic approach is most appropriate. A functional approach to linguistics views language in terms of what the speaker can do with it; it attempts to tie language to the social purposes for which it is used. The linguistic theory used throughout this thesis (systemic grammar) is a functional theory. It is therefore important that the grammars described here, and indeed the linguistic theory itself, not be judged by structural or generative criteria. The functional framework also means that there are several terms used in this thesis in an unorthodox manner. Systemic grammar, the concepts and terms, are discussed in Chapter 3.

1.3. Important assumptions

Having defined the context of this thesis, and explicitly stated that the linguistic coverage is a problem that will not be addressed, it must be pointed out that a vital assumption has been made here. Although the grammar implemented for this project is relatively large, it still has very limited coverage. An assumption has been made that systemic grammar, with enough work and development, can adequately describe the grammar of natural language for the purposes of text generation. A more precarious assumption has also been made, viz. that the semantic component of systemic theory is adequate. The justification for these assumptions is discussed in Section 9.2.

1.4. Specific overview

At this point a brief outline of the remaining chapters should be given. The thesis is roughly divided into three sections. The first three chapters (including this one) provide the background and introduce the relevant terms and concepts. The second three chapters form the core of the thesis--they put forward the original ideas. The final three chapters support and consolidate these ideas

by illustration, comparison and discussion.

The title of Chapter 2 is "Background I: AI problem solving." This should not be interpreted as background to the field in general, but rather as background to those problem-solving terms and concepts that play a major role in the remainder of the thesis. The primary purpose of this chapter is to introduce the key concept of "search" and some powerful knowledge-based search techniques.

Chapter 3 is also a background chapter; it provides the background to systemic grammar. Again the treatment is heavily biased toward the terms and concepts that are important later on. This chapter also provides a brief history of the development of systemic theory to vindicate some of the less orthodox ideas. Examples are provided to illustrate the key concepts.

Chapter 4 is the crux of the thesis. Here the special relationship between AI problem solving and systemic grammar is unfolded and developed into an approach to text generation that surmounts the problem of interfacing AI problem-solving methods and an established linguistic formalism. This "Systemic Linguistic Approach to Natural-language Generation" (SLANG) is then illustrated by working through an example in some detail.

Having presented SLANG, the new approach to text generation, and shown how it overcomes specific text-generation problems, the next chapter--Chapter 5--examines the approach from a broader perspective. The approach raises several theoretical issues concerning both scientific explanation and linguistic theory. Specific issues addressed here are the functional approach, the interface between the semantics and the grammar, and the differences between this linguistic approach and generative grammar.

One problem with systemic grammar is that it has never been given a formalization similar to that of more traditional grammars. This makes it difficult to provide a formal model for this approach to text generation. These problems are remedied in Chapter 6: "The formal model." One of the interesting side-effects of the

relationship between systemic grammar and AI problem solving is that it provides the basis for an almost traditional formalization of this functional grammar. This allows formal definitions to be given for "valid syntactic structure" and "derivation" and so on. A formal analysis of SLANG in terms of computational algorithms is also provided. Several lemmas concerning both the formalization of systemic grammar and the computational process are proven.

The text generation, problem-solving and linguistic ideas appearing in the the thesis have been implemented in a test system: SLANG-I. This implementation is described in detail in Chapter 7. The description is supplemented by a discussion of the limitations of the implementation and some thoughts on alternative implementations.

Chapter 8 compares SLANG with other recent approaches to text generation. A scheme for classifying text-generation systems is presented, and SLANG is shown to combine successfully the positive attributes of both the major text-generation classes.

Finally, Chapter 9 provides a summary of the thesis and some thoughts toward the future. Both the problems that may obstruct future work, and some potential extensions and offshoots, are examined.

Four appendices have been included at the end of the thesis. Appendix A, a supplement to Chapter 7, is a brief tutorial on OPS5--the implementation language of SLANG-I. Appendix B is a collection of sample texts produced by SLANG-I. Appendix C is a transcript of the grammar used in SLANG-I. Finally, Appendix D is a listing of the OPS5 and LISP code of the SLANG-I prototype text generator.

2. Background I: AI Problem Solving

This chapter will provide the background in AI problem solving necessary for the remainder of the thesis. The treatment of this topic here is not intended to provide a comprehensive understanding of all the issues in this large field; only the specific issues and perspectives relevant to the argument will be discussed. Indeed, this chapter should be read as part of the thesis argument.

The discussion will begin by presenting the underlying notion behind AI problem solving: search. The methods that have been used to perform search will be examined and compared, culminating with a preview of rule-based search. Finally, a synopsis of knowledge compilation issues is given.

2.1. Search

Problems in an AI context are often described in terms of search. The idea is that a problem solver has a number of possibilities available to it, and the difficulty is to search through the alternatives to find a solution to the problem. The exact nature of the alternatives depends on the type of task. Planning, for instance, involves alternative actions or operations that can be performed by the agent. The problem is to search through the combinations of sequences of operations to find one that satisfies some pre-set criteria. Similarly in a design application, the alternatives may be the various spatial arrangements of components that can potentially be used to build an object. The task is then to find some configuration of components that, again, satisfies some pre-set criteria.

The reason this involves search is that the various alternatives are not independent. A set of alternatives that would independently satisfy the solution criteria may be incompatible. For instance, two actions may be required to go before each other, or two components may require the same space. Thus a suitable computational representation for the alternatives must not only describe all the possibilities, but also their interdependencies.

2.1.1. Brute-force search

There are several approaches to searching for a solution to a problem. The simplest is just to start looking at all the possibilities, one by one, until a solution is found. This is called "brute-force" or "blind" search. Two examples of this method are depth-first search and breadth-first search (see Barr et al., 1981, pp. 38-40). This approach may be adequate for problems where there are only a small number of alternatives, but it is seriously inadequate for large problems.

2.1.2. Heuristic search

Often, during search, all the alternatives are not equally promising, given a particular set of solution criteria. An "evaluation function" can sometimes be found that can indicate preference for particular alternatives as the search proceeds. The search can then immediately focus on the most promising of the alternatives, ignoring the others--at least as long as the promise is sustained. The advantage of such "heuristic" search over brute-force search is that by rejecting unlikely alternatives, the solution is often found much sooner.

There are, however, some problems with this type of heuristic search. For many domains it is difficult to find a suitable evaluation function. Also, if the function is expensive to compute--in particular if it must take into consideration many complex interactions--the benefits of heuristic search are lost.

For instance (following Bundy, 1983, p. 54), suppose a robot needs to be at a certain position to satisfy the solution criteria. An evaluation function could be devised that would favour moves toward this position. Now suppose the robot is to collect an object and return to the original position. In this case the evaluation function would not work because the robot must move away from the desired position in order to collect the object. The function could perhaps be modified to take into account the distance between the robot and the object, but suppose more than one object is to be

collected. Even if a function could be written that takes everything into account, it would be so expensive to compute that it would be doing all the work instead of the heuristic search mechanism. Applying large amounts of knowledge to work out interactions is not a bad idea, but in this case the problem-solving process is no longer this kind of simple heuristic search.

2.1.3. Forward-chaining

One way to express problem-solving knowledge is in the form of rules. Rules can be expressed in several ways, but there is always a set of "conditions" and a set of "effects." Rule-based systems can reason from the conditions to the effects, or from the effects back to the conditions. These forms of reasoning are called forward-chaining and backward-chaining respectively. Forward-chaining will be examined in this section.

It was pointed out above that one of the reasons search is necessary is that there are complex interactions between the various sets of alternatives. Knowledge of these interactions can be used to guide the search. For instance there may be a rule that says:

```
IF alternative A has been chosen
   AND alternative C has been chosen,
THEN choose alternative X.
```

If in fact the problem solver has already decided to choose A and C then it can use this rule instead of using search to decide between X and other alternatives. Note that the problem solver can use chains of these rules to reason from an initial situation toward the solution--hence the term "forward-chaining." For instance, in a medical domain the knowledge base may contain rules like:

```
IF the patient has symptom X
   AND symptom Y
   AND symptom Z
THEN the patient has condition C.
```

IF the patient has condition C
AND is over 40 years old
THEN prescribe drug D.

Of course, the solution to complex problems may involve the construction of several interconnected chains of reasoning.

2.1.4. Goal-directed backward-chaining

Another form of rule-based reasoning involves chaining together rules starting from the solution--the goal--and working backwards. This is called goal-directed backward-chaining.

There may be a rule that achieves goal A but is conditional on two other (preferably simpler) problems, B and C, having been solved. The problem solver then sets B and C as subgoals and attempts to solve them recursively. Problem A has been "reduced" to problems B and C. This problem-solving strategy is thus called "problem reduction." If the problem solver is successful, the problems will eventually decompose into problems that can be solved directly by applying a rule whose conditions are already satisfied.

Backward-chaining uses rules very similar to those used for forward-chaining. In fact some systems (e.g. MYCIN, see Hasling et al., 1984) use the same rules for both forward- and backward-chaining. The inference engine simply looks in the knowledge base for a rule whose effects directly satisfy the goal. The rule is only applicable if the conditions are satisfied--so they are set as subgoals and solved recursively. Eventually the goals will be reduced to problems which can be solved directly or which are already solved as part of another problem.

The effectiveness of goal-directed backward-chaining depends on the interdependence of the rules. In the worst case many of the rules have disjunctive conditions, and several rules are applicable to each goal. In this case goal-directed backward-chaining degenerates into blind search because the problem solver does not know which rules and which subgoals result in a solution. In the best

case there will be no disjunctive entry conditions and only one rule will be applicable to each goal and subgoal. In this case the "search" is deterministic.

2.2. Compiled problem-solving knowledge

> [T]he quality true experts seem to possess that laymen do not is an ability to recognize large-scale patterns and jump quickly to reasonable hypotheses. Expert behaviour seems to demand that blind search through large numbers of hypotheses be avoided in favour of quick elimination of many possibilities in each inferential move.
>
> High-level macromoves that allow large amounts of ground to be covered in each step are a key feature of all the expert systems that have been built to date. (Brachman et al., 1983, pp. 44)

> Compiled knowledge. Knowledge that encodes rules of inference in which implied chains of reasoning are suppressed for the sake of efficiency (Brownston et al., 1985, Glossary).

A topic that deserves substantial attention here is "compiled" knowledge. This is of interest to AI because it is this kind of knowledge--together with techniques such as forward- and backward-chaining--that has led to the success of AI problem solving in expert systems. It is currently of particular interest because of the recent research into giving expert systems the ability to reason from first principles using deep as opposed to compiled knowledge (see Chandrasekaran and Mittal, 1984). The topic of compiled knowledge is also important for discussions in later chapters.

2.2.1. Sources of compiled knowledge

Much of the compiled knowledge used to date has been acquired in that form from human experts, or compiled by hand. There are also limited means of automatically compiling knowledge. Both of these sources are important in later chapters, so each will be briefly discussed in this introduction.

2.2.1.1. Precompiled knowledge and compilation by hand

Most compiled knowledge in the problem-solving literature was simply acquired in that form during the knowledge acquisition phase of building an expert system--the human expert uses this sort of compiled knowledge to solve problems in his domain of expertise. Another possibility is that the problem solving in some specific cases is too slow. The knowledge engineer may decide to add some high-level specific rules that replace the long reasoning process in these cases.

2.2.1.2. Automatic knowledge compilation

While compiled knowledge is desirable for reasons of efficiency, the rules are often awkward and difficult to understand, modify and so on. A useful technique to avoid these problems is to have the knowledge engineer work with knowledge of a certain grain size (level of compilation, see Hobbs, 1985), and then automatically compile the knowledge to a larger grain size.

Probably the best known example of automatic knowledge compilation is the MACROP (macro operator) facility in STRIPS (see Bundy, 1983, pp. 60-62; Barr et al., 1981, pp. 131-134). The basic idea is that after constructing a plan to achieve some task, the plan is generalized (by replacing specific tokens with variables where possible) and saved for future use. The more of these MACROPS that have been saved, the less work the planner has to do on the fine details--the grain size of the planner's work increases.

Another type of automatic knowledge compilation is to have a preprocessor that takes a rule-base and compiles it into larger, more efficient rules. This technique, like the construction of MACROPS, has the advantage that the knowledge engineer does not have to write, modify, understand etc. rules with too large a granularity (see Brownston et al, 1985, pp. 263-264).

2.2.2. Reasoning from first principles

After the initial success of expert systems that relied on compiled knowledge, there has recently been an interest in "reasoning from first principles." This type of reasoning is used to supplement the reasoning with compiled knowledge, primarily for reasons of robustness and explanation.

> The principle quality that general knowledge and inferential ability produces, over and above what expert rules do is robustness. As new, unanticipated patterns crop up, inflexible, compiled solutions fail. General problem-solving abilities allow a more graceful degradation at the outer edges of domain knowledge--a kind of conceptual extrapolation--as well as permit interpolation between high-level rules that are not complete within the domain....
>
> It should be noted that this type of knowledge is essentially the antithesis of high-level macro-move expertise. It is knowledge that is explicitly not compiled, so that it may support general inferential procedures. Applying knowledge with general methods, however, is inevitably slower than using multi-step inferential rules. (Hayes-Roth et al., 1983, p. 46)

This "reasoning from first principles" has also been advocated for providing explanations of high-level reasoning.

> Explanation in expert systems is usually associated with some form of tracing of rules that fire during the course of a problem-solving session. This is about the closest to real explanation that today's systems can come, given the fact that their knowledge is represented almost exclusively as high-level rules. However, a satisfactory explanation of how a conclusion was derived demands an ability to connect the inference steps with fundamental domain principles as justifications.... Each high-level macromove can be justified only by recourse to the basic principles that make it sound--the rule cannot be its own justification. (Brachman et al., 1983, p. 48)

2.3. Summary

The purpose of this chapter has been to provide the background necessary to understand the AI problem-solving terms and concepts appearing later in the thesis.

The concept of search was introduced first. Then some methods for performing search were surveyed. Of particular note were the knowledge-based techniques of forward-chaining and goal-directed backward-chaining.

The issue of the compilation of the knowledge used by problem-solving methods was then examined. The important concept here was that problem-solving knowledge can be expressed at different levels of compilation or granularity. The knowledge at higher levels of compilation can be more efficient, and knowledge at lower levels of compilation is robust and allows detailed explanations. Knowledge-based problem-solving systems can contain knowledge of different degrees of compilation, and may have mechanisms for automatically raising the level of compilation.

3. Background II: Systemic Grammar

Any work on text generation must give an account of the linguistic formalism--adopted or created--on which the generation process operates. This chapter is an introduction to the linguistic formalism adopted here--systemic grammar. The linguistic representation plays a particularly important role in this thesis. Indeed, an understanding of many of the computational text-generation ideas requires an understanding of the underlying concepts in systemic theory.

This introduction to systemic grammar begins with a short history focused on the major contributors: Malinowski, Firth and Halliday. Then the goals or aims of systemic grammar are outlined. Some of the concepts from systemic theory which are most relevant to this thesis are then discussed in detail. Finally, descriptions of the stratification of systemic grammar, and in particular of the semantic stratum, are given.

3.1. History

3.1.1. Malinowski

The origins of systemic linguistics clearly lie in the work of the anthropologist Bronislaw Malinowski (e.g. 1923). From Malinowski come two ideas that have had a profound influence on systemic theory. The first is the observation of the inseparability of language and its social and cultural context (Whorf must also be credited as an influence on this point--Kress, 1976, pp. ix-x). Malinowski argued that language could only be viewed and explained with reference to the social and cultural milieu. It is important to note the sharp contrast between this starting point of systemic linguistics and the starting point of the structural/formal tradition: that language is a self-contained system (ibid., p. viii). Most importantly here, Malinowski provided the idea of "context of situation"--an abstract description of the contextual factors influencing an utterance.

The second important idea from Malinowski is that language is "functional"--it is used to perform certain functions in society. Of particular note is his grouping of the functions of a particular language into broad but culturally dependent categories. For instance, one of the functions of language in the Polynesian societies studied by Malinowski is the "magical function" where language is used to control the environment (ibid., p. viii).

Malinowski's influence remained unmistakable as his ideas were refined and developed by others. The first step in the refinement process was to transfer the thinking of the anthropologist into a linguistic framework.

3.1.2. Firth

It was the linguist J.R.Firth who took Malinowski's ideas and adapted them so they could fit into a linguistic theory. In particular he accepted the close relationship between language and society put forward by the anthropologist.

One key notion in Firth's work was the concept of "system" (from which systemic grammar eventually took its name)--a set of linguistic choices in a specific linguistic context (ibid., p. xiii). Firth's emphasis on differentiating (according to de Saussure's dichotomy) this "paradigmatic" (system-based) description and the "syntagmatic" (structure-based) description set him apart from the Bloomfieldian tradition (Halliday and Martin, 1981, p. 19).

Firth realized that words or sentences could not just be related directly to a general context, but rather the context had to be divided up into different levels--as he said, like "breaking white light into a spectrum" (Monaghan, 1979, p. 185). Thus the phonological choices must be made in a phonological context, grammatical choices must be made in a grammatical context and so on.

Another key observation was that the general situation types described by Malinowski resulted in a "multiplicity of languages" within a language as a whole (Kress, 1976, p. xiv). This insight

later led Halliday to the important concept of register (see Section 3.5).

3.1.3. Halliday

Though Malinowski and Firth had made many important observations and insights, a complete linguistic theory had still not been developed. Halliday took this previous work and extended and refined it to produce the theory of systemic grammar (originally presented in Halliday, 1961).

Halliday adopted Firth's emphasis on paradigmatic description, and took it even further, to the point where the paradigmatic description clearly dominates the syntagmatic description. Another extension from Firth was the presentation of a coherent set of categories which could be related to each other at the interface between different levels of context (e.g. between the grammatical and phonological levels. Kress, 1976, p. xv). Later work included a grammar notation--system networks--and the work on register stemming from Firth's "multiplicity of languages" (e.g. Halliday, 1978). The remainder of this chapter will explore in some detail Halliday's theory of systemic grammar.

3.2. The goals of systemic grammar

The previous sections have shown that the origins of systemic grammar differ significantly from those of the currently dominant school of linguistics. The roots of systemic grammar

> ... were in anthropology and sociology, not in mathematics or formal logic. The questions that motivated its development were not those of grammaticality or the acquisition of linguistic competence, but those of language as a social activity: What are the social functions of language? How does language fulfill these social functions? How does language work? (Winograd, 1983, p. 273)

Some of the relevant goals of systemic grammar result from the historical interests introduced in the previous sections. The first of these is the goal of describing the function of language. There

are several levels at which this description must be made. On one hand there is what might be called the semantic function--an utterance functions as a question or a statement, or part of an utterance may identify the performer of an action or what is being talked about. On the other hand there is what might be called the syntactic function--the "subject" of a clause, the "head" of a nominal-group and so on. Thus one goal of systemic grammar is to capture the subtle relationship between the semantic function, the syntactic function and the form itself (ibid., p. 277).

Another important goal in systemic grammar, as in other grammars, is the description of the constituent structure of language. The concept of constituency is the same in systemic grammar as in more traditional grammars, but the goal of describing the various functional aspects of language will force a somewhat different approach to this topic.

Finally, and of primary significance for the present work, an important goal in systemic grammar is the classification at all levels of linguistic alternatives. The classification of both social situations and linguistic forms "plays a major theoretical role in systemic grammar" (Winograd, 1983, p. 276). A result of this classification is that a systemic grammar embodies a paradigmatic description of all the alternatives in meaning and in form available to the speaker.

3.3. Important concepts in systemic grammar

A brief look at the history and goals of systemic grammar has been presented, and an introduction to the theory itself will now be given. This will not be a thorough linguistic treatment, but will attempt to provide some insight into the concepts from systemic grammar that play a significant role in the remainder of this thesis. A good general overview of systemic grammar can be found in (Winograd, 1983, Chapter 6).

3.3.1. Feature

Probably the best starting point is the notion of a "feature." One of the primary goals of systemic grammar is classification (see Section 3.2 above) and a feature can be defined as the name of a class (Halliday and Martin, 1981, Glossary). Some features of the clause (classes to which a clause may belong), are declarative, finite, benefactive, negative, interrogative, positive and so on.

Now it should be apparent that these features are not all independent. If a clause has the feature declarative then it cannot also have the feature interrogative. Similarly if a clause is negative then it cannot also be positive. This leads to the concept of "system."

3.3.2. System

A system is a mutually exclusive set of classes (or features) and thus represents a choice or "potential." This description of language in terms of choices is the "paradigmatic" description mentioned above. Note that this is important from the point of view of classification and information theory--if a clause is labelled as declarative it also means that the clause is not interrogative.

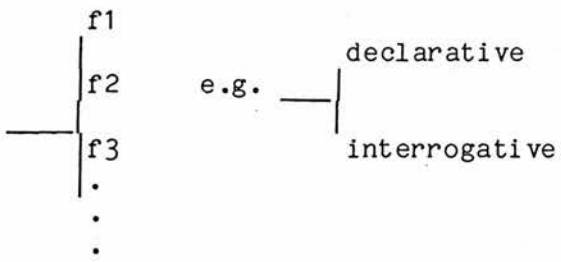
The next step is to observe that a particular choice is not always applicable--e.g. a linguistic item is not always either declarative or interrogative. Thus some sort of context must be introduced to determine which choices are relevant when. For Firth, the context was a structural one--the relevant choices were directly dependent on the structure of the linguistic item. Halliday, however, made the radical step of defining the context in terms of other choices. For instance the choice between declarative and interrogative is only appropriate if the clause is indicative as opposed to imperative.

Often a choice will depend on a logical combination of features instead of on just one. In any case, the features that must be present for a system to be appropriate are called the "entry

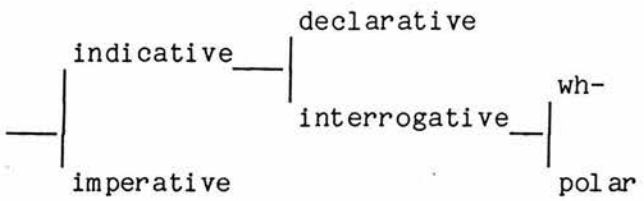
conditions" of the system. The system and entry condition relationships can be illustrated by drawing a "system network."

3.3.3. System network

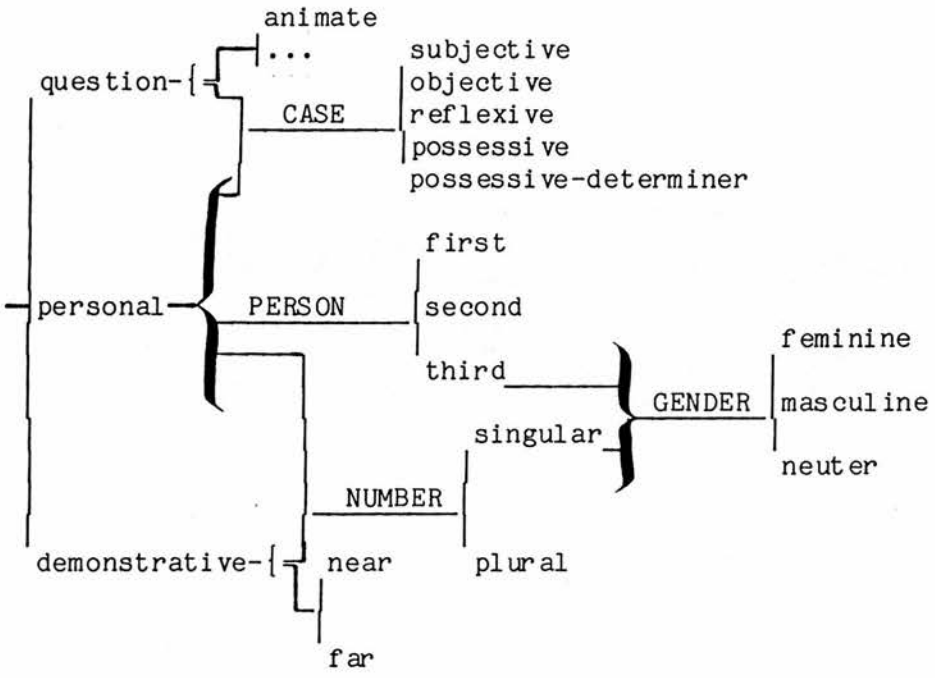
System networks display graphically the relationships between features in the grammar. A system is illustrated by a "T" intersection (representing a choice between two or more features):



Entry conditions are illustrated by simply drawing lines from the entry conditions to the system:



If several features are involved in entry condition relations--either a feature acts as an entry condition to several systems or a particular system has several entry conditions--this is illustrated with curly brackets, "{" and "}" respectively. Disjunctive (not necessarily exclusive) entry conditions are represented by a square bracket ("T" merge) "]-".



System networks for English pronouns

Figure 3.1 (from Winograd, 1983, p. 293)

Consider Figure 3.1. Here there is a variety of complex relationships between features. Features are in lower case; system labels are in upper case and are merely for documentation. The feature question is the sole entry condition for the system containing animate and is a disjunctive entry condition for the CASE system. The feature personal is the entry condition for the PERSON system and a disjunctive entry condition for the CASE and NUMBER systems. The features third and singular must both be chosen if the system GENDER is to be relevant.

In addition to features which are terms in systems, there are features--called "gates"-- which are simply dependent on some combination of other features, without choice. These could be thought of as degenerate systems with only one feature. The entry conditions of gates are represented in exactly the same way as those of systems.

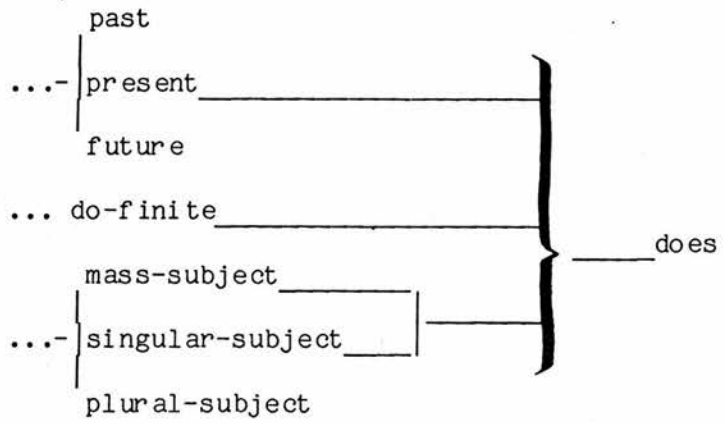


Figure 3.2.
A gate from a clause network (Mann/Halliday).

In Figure 3.2, if the features present and do-finite have been chosen, and either mass-subject or singular-subject have been chosen, then the feature does is chosen as well--there is no choice here. The entry conditions to gates may be terms in systems or other gates: in Figure 3.2 present, mass-subject, and singular-subject are terms in systems, while do-finite is another gate.

Excerpts from the clause system network, and the pronoun system network (actually part of a larger noun network) have been presented. System networks are required for constituents such as the prepositional phrase, the clause-complex (roughly corresponding to a sentence) and so on. "The grammar itself thus takes the form of a series of system networks, where each network represents the choices available to a given constituent type,..." (deJoia and Stenton, 1980, #685). Although there are clearly structural relationships between the types of constituents, the system networks allow a complete paradigmatic description--in terms of feature choices--to be given for any particular constituent without referring to its substructure at all.

3.3.4. Delicacy

> As with any classification system, a system network for syntactic objects can go to varying levels of detail. In biology, an organism that is assigned a species feature is more precisely described than one assigned only to a

> family or genus. The more precise the classification, the more information is available about the object. In systemic grammar, this scale of precision is called delicacy.... (Winograd, 1983, p. 296)

Delicacy applies to features and systems, and is clearly illustrated in system networks. Generally speaking, system networks increase in delicacy from left to right. Some of the delicacy relations in Figure 3.1 are: the feature masculine is more delicate than the feature personal; the system GENDER is more delicate than the system NUMBER; the features subjective and objective are of equal delicacy; and the system whose terms are question, personal and demonstrative is the least delicate system.

3.3.5. Functional analysis

Another important concept in systemic linguistics is the idea of "function." Functional analysis in systemic grammar consists of more than just labelling linguistic items with terms like "Subject" and "Agent." The theory provides for analysis of several functional dimensions simultaneously, and indeed a large part of the linguistic description consists of relating these analyses.

	This gazebo	was built	by Sir Christopher
MOOD	Subject	Predicator	Adjunct
TRANSITIVITY	Goal	Action	Actor
THEME	Theme	Rheme	

Figure 3.3. (Winograd, 1983, p.283)

Figure 3.3 shows three functional analyses of the same clause. Each function is associated with one type of analysis. The function Actor, for instance, is always used in the analysis of transitivity, and the function Subject is always used in the analysis of mood. The different analyses are related by "conflating" functions from different analyses (e.g. Subject, Goal and Theme are all interpretations of "this gazebo" above).

A consequence of this multidimensional functional treatment is

that there are in fact several constituent analyses associated with a linguistic item--one for each different functional analysis. This thesis will use four functional analyses: transitivity, ergativity, mood and theme.

3.3.5.1. Transitivity

The analysis of transitivity (see Halliday, 1985, p. 101-144; Winograd, 1983, pp. 497-504) is an analysis of "process." The process per se is represented by the function Process (realized by "built" in Figure 3.3). The remainder of the set of functions depends on the nature of the process.

The first type of process is "material" (Halliday, 1985, p. 102-106), a process of "doing" or "creating." The primary functions are Actor and Goal. A Beneficiary appears in the case of benefactive processes.

Jack	gave	the book	to	Janet
Actor	Process	Goal		Beneficiary

the apples	were	eaten	by	Jack
Goal		Process		Actor

If the process is "mental" (ibid., 106-112) then the functions are the Process, the Senser, and the Phenomenon.

I	like	cheeseburgers
Senser	Process	Phenomenon

cannons	hurt	my ears
Phenomenon	Process	Senser

If the process is "verbal" (ibid., p. 129) then the functions are the Process, the Sayer and what is usually an embedded clause,

Beta.

I	said	it was cold
Sayer	Process	Beta

Finally, the process may be "relational" (ibid., p. 112-128). There are two kinds of relational process, each with its own set of functions. There are "attributive" processes which involve an Attribute and a Carrier.

the book	is	pathetic
Carrier	Process	Attribute

my dog	has	fleas
Carrier	Process	Attribute

There are also the "identifying" processes which involve the functions Identifier and Identified.

Jack	is	the vice-president
Identified	Process	Identifier

the fleas	are	his
Identified	Process	Identifier

3.3.5.2. Ergativity

Halliday (1985, pp. 144-157) has argued that transitivity is no longer as important an analysis of English as it once was. The idea of transitivity is that there is a process and an Actor, and the question is whether or not the process extends beyond the Actor to something else (the Goal) (ibid., p. 145).

a) the gun fired (intransitive)

b) the gun fired the bullet (transitive)

Here "the gun" is the Actor in each case, and b) is transitive because the bullet is the Goal. However, according to Halliday (ibid.) the majority of high-frequency verbs that can be either transitive or intransitive, yield pairs such as:

a) the glass broke (intransitive)

b) the singer broke the glass (transitive)

Here the relationship isn't really transitivity at all. The process of the glass breaking in a) does not extend to the singer in b) as did the firing of the gun in the previous example. The distinction being made in this case is whether the process was caused or not--ergative or non-ergative processes respectively. The functions used for the ergative analysis are the Process, the Agent (called the Causer in some of the earlier literature), the Medium (earlier called the Affected), and perhaps the Beneficiary.

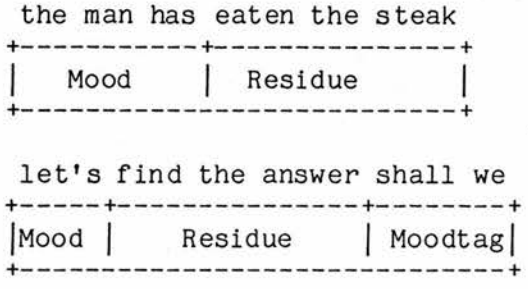
	the glass	broke
ERG	Medium	Process
TRANS	Actor	Process

	the singer	broke	the glass
ERG	Agent	Process	Medium
TRANS	Actor	Process	Goal

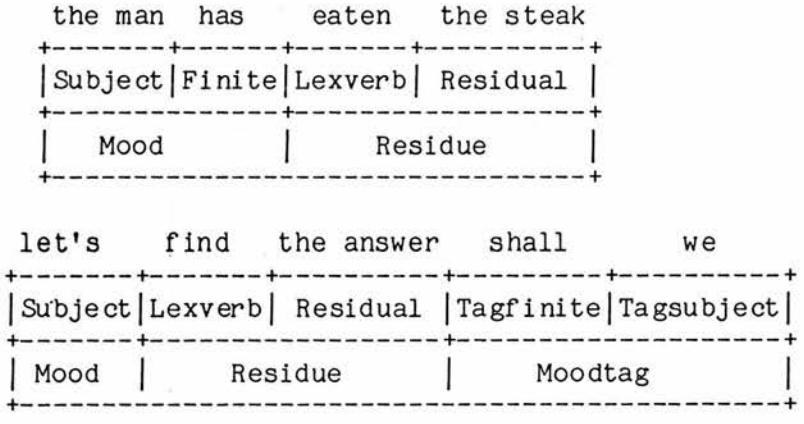
Notice that the Medium is constant in the above examples, whereas the Actor shifts in the transitivity analyses. Both the transitivity and ergativity analyses, as well as their interaction (sometimes the Medium is conflated with the Actor, sometimes with the Goal) are useful.

3.3.5.3. Mood

The functional analysis of mood (Halliday, 1985, pp. 68-100) is slightly more complex than that of either transitivity or ergativity. This is because there is more than one level of analysis. At the top level the functions used are: Mood, Residue and, optionally, Moodtag.



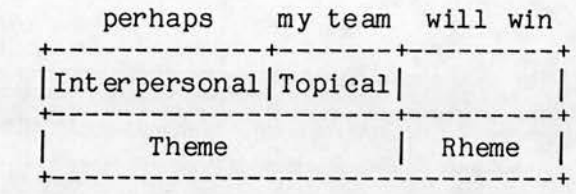
Each of these functions is divided or "expanded" into a number of subfunctions. The Mood is expanded into the Subject and the Finite, the Residue is expanded into the Lexverb (this differs from much of the systemic literature) and the Residual, and the Moodtag is expanded into the Tagsubject and the Tagfinite.

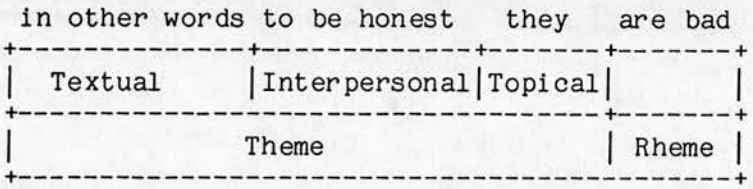


3.3.5.4. Theme

Like the analysis of mood, the analysis of theme (ibid., pp. 38-67) involves several layers of functions. At the top layer are the functions Theme and Rheme. The Rheme is not expanded further, but the Theme is expanded into the Textual, the Interpersonal, and the Topical. These are expanded further in (Halliday, 1985) but the

further subdivisions are not used here. The Topical is usually conflated with the Subject, the Interpersonal is a modal adjunct (ibid., p. 50), and the Textual is a conjunction or conjunctive adjunct (ibid., and especially Halliday and Hasan, 1981, Chapter 5).





3.3.5.5. Information

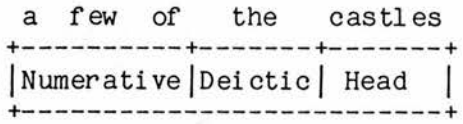
Another functional analysis is important when working with speech. This is the analysis of "information structure" and involves the functions Given and New (Halliday, 1985, pp. 274-251; Winograd, 1983, pp. 505-506). The portion of a "tone group" (often a clause) conveying information already possessed by the hearer functions as Given; the portion of the tone group conveying information new to the hearer functions as New. Information analysis is germane to issues of stress, intonation and word order.

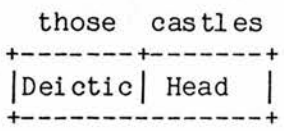
Since the information analysis is largely concerned with speech issues, it has been excluded from this work to avoid the added complexity.

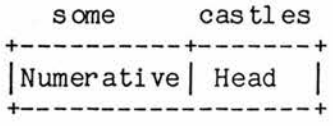
3.3.5.6. Functional analysis for groups

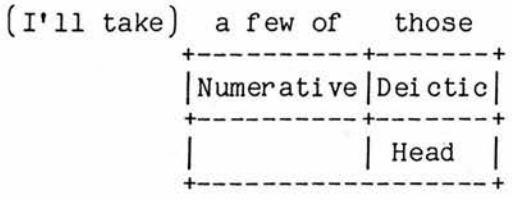
The functional analysis of the group is much less complex than that of the clause. Although relatively complex group analyses are provided in (Halliday, 1985, p. 159-175), the simpler analyses given in (Halliday, 1976a, p. 131-135) are used here. The only substantial group network used is the nominal-group, because the verbal-

group is treated in the analysis of the clause. The functions appearing in the analysis of the nominal-group are the Numerative (a quantifier), the Deictic (usually a determiner), and the Head (often a noun, pronoun, substitute etc., but may be conflated with either of the other two functions):









3.3.6. Rank

Although the emphasis in systemic grammar is primarily on the functional issues of language, it still must relate this function to structure. This requires a structural analysis that is similar to that found in the traditional "immediate constituent" grammars, but that is also consistent with all of the different functional analyses. The deep, narrow trees (where each node has a small number of constituents) produced by immediate constituent grammars typically will conflict with at least one of the functional analyses.

Systemic grammar therefore adopts an approach called "minimal bracketing" (where constituents are grouped together in a separate level of structure only when absolutely necessary, see deJoia and

Stenton, 1980, #3). In fact there are only a small, fixed number of groupings called units: the "clause-complex," the "clause," the "group"/"phrase," the "word," and the "morpheme." The minimal bracketing of a "rank grammar" has a significant effect on the constituent analysis: the constituent trees are short and bushy rather than long and narrow.

Clearly there is a hierarchical relationship between the various units. The constituents of a clause, for instance, will usually be groups and words, while the constituents of a group will tend to be words. For this reason the relationship between these various units is called rank. Since all constituents in systemic grammar are at one of these ranks, systemic grammar is called a "rank grammar" (see deJoia and Stenton, 1980, #608, #609). Note that the top and bottom ranks (clause-complex and morpheme) will not be used in this work.

A constituent normally realized by units at a particular rank may occasionally be realized by a unit of a higher rank. For instance the Deictic in a nominal-group is normally an item at the word rank (e.g. "that" in "that hat"). In the case of a possessive determiner, however, the Deictic may be a nominal-group acting as a word (e.g. "the elephant's trunk"). This is called "rankshifting."

Although "rank" and "unit" are important concepts in systemic theory, they do not appear to play a significant formal role (see Chapter 6), contrary to what one might expect given their prominence in the systemic literature (Halliday, 1961, for instance). This is largely due to the phenomenon of "rankshifting," since no formal restriction can be placed on the rank of constituents.

It is important to distinguish between "rank" and "delicacy." It is easy to confuse these two scales of abstraction, but in fact they are orthogonal. The feature nominal-group is not more delicate than the feature clause; they are each the least delicate features at their respective ranks. Starting at the feature clause, and increasing in delicacy to finite to indicative to interrogative, the description is not moving toward smaller constituents, but to finer

distinctions between classes of clauses.

3.3.7. Realization rules

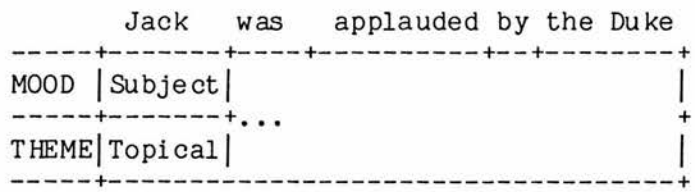
The features and system networks have been introduced, as have been the ideas of functional analysis and rank. But there is a gap left to be filled between the features and system networks on the one hand, and the functional analysis and constituent structure on the other. This gap is filled by the "realization rules" attached to the features in the grammar.

The realization rules can be regarded as specifying the structural implications of the feature to which they are attached. Elements of structure are represented in realization rules by their function (e.g. Subject, Agent). The set of functions described in Section 3.3.5 is fairly standard, but unfortunately the realization relationships vary from source to source, and there seems to be no standard notation for even the widely-used ones.

The notation used in this thesis is taken from (Mann/Halliday). An additional convention of enclosing realization rules in parentheses has been introduced. Some examples of the various realization rules and their associated features will now be presented.

3.3.7.1. Conflation (/)

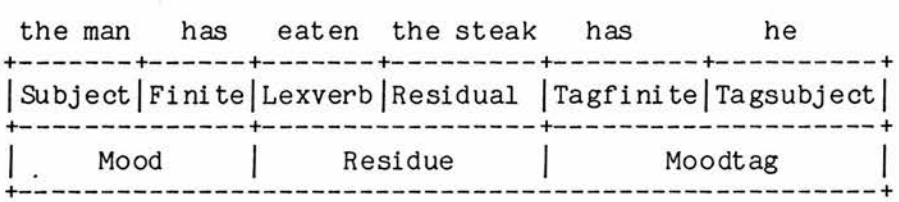
A realization relationship that seems to be used universally in systemic grammars is "conflation" (the symbol is "/" in Mann/Halliday but "=" in Winograd, 1983, p. 305). This states that the same linguistic item realizes more than one function. For instance the feature unmarked-declarative-theme has the realization rule (Subject / Topical), as in "Jack was applauded by the Duke," where "Jack" is functioning as both the Subject and Topical:



3.3.7.2. Expansion (())

The "expansion" realization rule takes two arguments: a function to be divided into subfunctions, and one of the subfunctions. For instance, in Section 3.3.5.3 Mood was expanded into Subject and Finite--this is written as the two realization rules (Mood(Subject)) and (Mood(Finite)), attached to the features indicative and finite respectively. Similarly, (Theme(Topical)) is attached to topical-inserted, and so on.

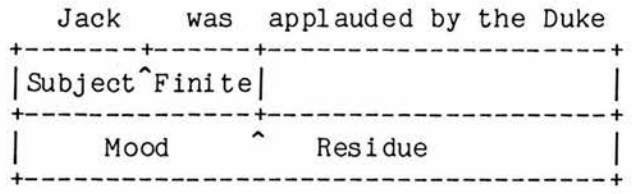
Expansion is indicated in the structure diagrams where there are two levels of the same analysis, and one of the functions in the bottom row spans exactly the same distance as two or more functions in the top row. The expansions (Mood(Subject)), (Mood(Finite)), (Residue(Lexverb)), (Residue(Residual)), (Moodtag(Tagfinite)) and (Moodtag(Tagsubject)) are drawn:



3.3.7.3. Adjacency (^)

Another realization relationship is "adjacency." This states that the linguistic items realizing two particular functions are adjacent in the structure. Consider the feature declarative, which has the realization rule (Subject ^ Finite), where "^" is the symbol for adjacency. For instance, in "Jack was applauded by the Duke," "Jack" is the Subject, and "was" is the Finite element. The feature finite has the realization rule (Mood ^ Residue). In the same

example, which is also finite, "Jack was" is the Mood, and "applauded by the Duke" is the Residue.

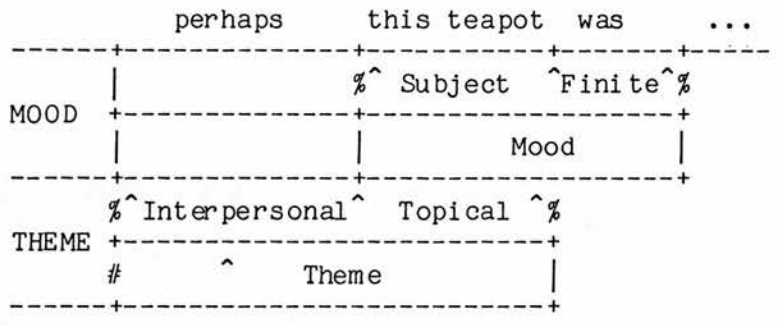


Some other grammars use a realization relation which merely indicates that one of the functions appear after (as opposed to immediately after) the other (e.g. Winograd, 1983, p. 305; Mann et al., 1983). This can be used instead of, or as well as, adjacency.

A special case of adjacency is that in which an item is a leftmost or rightmost constituent, and therefore adjacent to the boundary. The feature clause has the realization rule (# ^ Theme) indicating that in all clauses the Theme is at the beginning. This thesis treats the boundary symbols as quasi-functions that appear in adjacency statements like other functions. Mann et al. (1983) have opted to have special realization relationships called "order-at-front" and "order-at-back" which take one real function as an argument. These are simply two notational variants on the same theme.

Since it is convenient to be able to state, for instance, that Subject is the leftmost subfunction of Mood, a new symbol has been introduced to denote the boundary of an expanded function: %. The realization rule (% ^ Subject) indicates that Subject is the leftmost subfunction of some expanded function. This is not ambiguous since a function can only be a subfunction of at most one function--though it can be associated with other expanded functions via conflation. For instance Topical is a subfunction only of Theme, and Subject is a subfunction only of Mood, and the Subject and the Topical may be conflated; but there can be no realization rules (Mood(Topical)) or (Theme(Subject)). Expansion, together with conflation, allows very complex structures to be specified. For instance, (Mood(Subject)), (Mood(Finite)), (% ^ Subject), (Finite ^ %) (Theme(Interpersonal)), (Theme(Topical)), (% ^ Interpersonal), (Topical ^ %), (# ^ Theme) and (Subject / Topical) constrain the

first three items in the clause to be the Interpersonal, the Topical/Subject followed by the Finite:



3.3.7.4. Lexify (=)

The grammar may require a function to be realized by a particular lexical item. This is indicated by the realization relationship "lexify." This is most often found at the word rank, but is also found at the clause and nominal-group ranks.

In fact lexify is not used in the clause network (Mann/Halliday) but was adopted from (Mann et al., 1983, see p. 25) for convenience. For instance the feature speaker-subject was given the realization rule (Subject = I) as in:

3.3.7.5. Preselection (:)

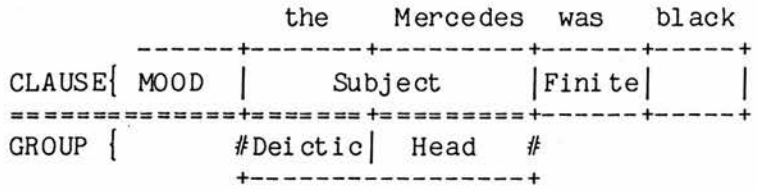
A realization rule that is particularly important in this thesis is "preselection" (the symbol ":" is used in Mann/Halliday, but it is called "classification" with the symbol "/" in Winograd, 1983, p. 305). This is the form of realization used to interface the different system networks. Sometimes classification at the clause rank, for instance, implies classification for its constituents at other ranks. A preselection classifies a linguistic item (identified by a function) by selecting a feature for that item from

a network representing a lower level of classification. For instance, the feature speaker-subject mentioned above has the realization rules (Finite : !first-person) and (Finite : !v-singular) which preselect, from the verb network at the word rank, the features classifying the Finite as a first person singular verb. The grammar described in (Mann et al, 1983) uses the symbol "!" instead of ":" when preselecting lexical features because that grammar has no networks at the word rank (e.g. (Finite ! pastform) ibid., p. 45). Even though the grammar used here does have word rank networks, it is useful to distinguish preselections from the word rank to avoid confusion where it is not clear to which rank a feature belongs (e.g. it may not be clear if the feature name singular has been used in the nominal-group network or the noun network). Thus in the grammar described throughout the present work, the symbol ":" is used for all preselections, but features at the word rank are prefixed with a "!" (!singular as opposed to singular). This is purely a notational convention to aid the reader; there is no linguistic or computational significance.

This (Function : feature) notation is fine so long as the feature applies to the constituent immediately below that represented by the function in the constituent tree. In the case of Finite, it is realized by a verb so there is no problem preselecting the verb features !first-person and !v-singular.

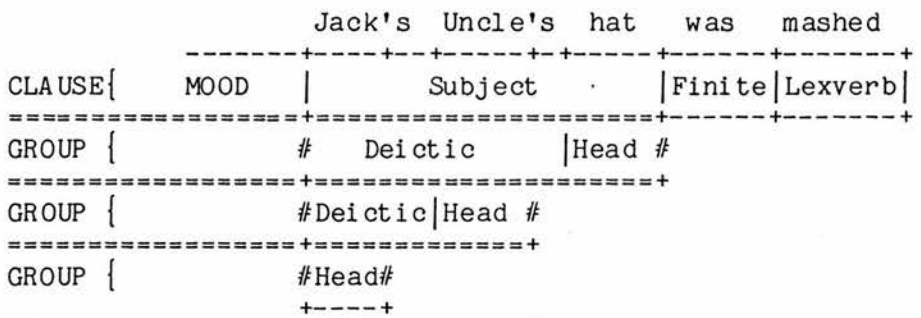
However, consider the case of the feature proper-subject in the clause network (Mann/Halliday). The problem is that the Subject will be realized by a nominal-group, and what is really needed is for the feature !proper to be preselected for the Head of that nominal-group--not for the nominal-group itself. This cannot be done with the notation currently in the systemic literature, short of introducing a special feature (e.g. proper) at the group rank which itself has the realization rule (Head : !proper). This intermediate feature addition has been avoided here by using a "path notation" for preselection realization rules. Instead of just giving the single function, a whole path of functions are specified, separated by the symbol "<" (symbolizing the constituent tree). The

feature proper-subject has the realization rule (Subject<Head : !proper).*



The group analysis is separated from the rest of the diagram by a double line to indicate that it is not part of the clause analyses.

Paths can easily be represented with and read from a structure diagram. For instance, in the clause "Jack's Uncle's hat was mashed", where there are several embedded groups:



Jack's Uncle's hat	is the Subject
Jack's Uncle's	is the Subject<Deictic
Jack's	is the Subject<Deictic<Deictic
Jack	is the Subject<Deictic<Deictic<Head
Uncle	is the Subject<Deictic<Head
hat	is the Subject<Head

* Although this added notation may not appear to be justified by the few cases for which it is needed, it is essential for other reasons discussed in Section 3.4.4.

3.3.7.6. No insertion/inclusion (+)

One type of realization rule which is almost universal in the systemic literature but not used here is "insertion" (Mann et al., 1983, p. 25), also called "inclusion" (Winograd, 1983, p. 305) and represented by the symbol "+". For instance, the feature finite (Mann/Halliday) has the realization rule (+ Finite) meaning that finite clauses have Finite elements. The feature determined in a nominal-group network may have the realization rule (+ Deictic) meaning that determined nominal-groups have an element functioning as Deictic.

This type of realization rule has not been implemented here because in any grammar detailed enough to be used in an automatic text-generation system, the functions that are inserted will always appear in other realization relationships (at least when given the set of relationships outlined above). Therefore the insertion statements are at least technically redundant. It could perhaps be argued that it is useful to provide insertion statements for linguistic clarity, but the author's experience has not indicated this (e.g. Note in Winograd, 1983, p. 305 that inclusion is almost always combined with another realization relationship in the same rule).

3.3.8. The metafunctions

Following Malinowski's observation that language functions can be grouped into abstract categories, Halliday has identified three general "metafunctions" in adult language.* These provide a valuable conceptual grouping, but like "rank" do not play a formal role in the model.

---

* Halliday's work concerning the functional aspects of the language of young children reveals a larger number of less developed "macro-functions" (e.g. Halliday, 1978, pp. 50, 55-56, and especially 121). There may be some confusion because the term "macro-function" was used in earlier writings (e.g. Halliday, 1973) to also refer to what are now called metafunctions.

3.3.8.1. The ideational metafunction

The ideational metafunction is language functioning to represent the "world" in general--processes, events, actions, objects etc., as well as logical relationships between them (Halliday, 1978, p. 21).

3.3.8.2. The interpersonal metafunction

The interpersonal metafunction is language functioning to express roles of the speaker in the discourse. The speaker is communicating: what is being talked about, the relationship with the hearer (e.g. contradicting, supporting), how strongly the text is believed, whether or not the speaker is happy about what is being said, and so on (ibid.).

3.3.8.3. The textual metafunction

The textual metafunction of language is to organize the text in such a way that it is internally cohesive, and fits into both the larger discourse and the social situation in general. In other words it ensures that the text is relevant and coherent (ibid.).

3.3.8.4. Metafunction and linguistic description

Although the metafunctions can be correlated with the different functional analyses, in keeping with the spirit of systemic grammar the metafunctions have their basis in the paradigmatic description. Looking at system networks of natural languages, Halliday noted that there tends to be a high degree of interdependence among some groups of features and relatively low interdependence between these groups. The groups of features correspond to the three metafunctions mentioned above.

> In origin ..., the concept of metafunction is an empirical claim about the paradigmatic organization of the clause systems in English. (Martin, 1984)

Halliday claims (1978, p. 21-22) that these metafunctions are

common to all adult natural language (not just English). Presumably other languages may have others, such as the "magical" metafunction observed by Malinowski in the Polynesian languages he encountered.

The functional analyses presented above are correlated with the three metafunctions. The realization rules attached to ideational features in general specify the transitivity and ergativity analyses, the realization rules attached to interpersonal features in general specify the analysis of mood (as illustrated by Figure 3.4), and the realization rules attached to textual features in general specify the theme analysis.

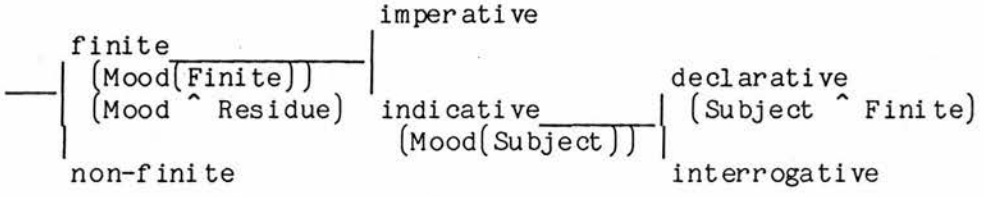
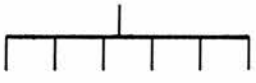


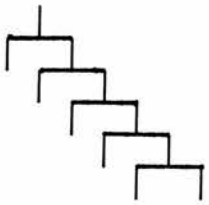
Figure 3.4
An excerpt from the interpersonal section of the clause network.

3.3.9. Recursive systems

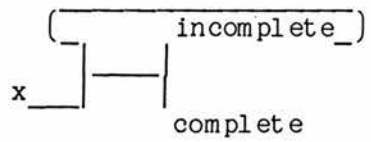
The multi-dimensional functional analysis and the principle of minimal bracketing have led to a serious problem with the system network notation. The advantages of minimal bracketing are often illustrated using the example of "parataxis." Parataxis is simply a logical combination of items of the same rank forming a list (see Halliday and Martin, 1981, Glossary--e.g. "John and Bill and Mary," "the red one, the blue one, or the black one," this list of examples and so on). The minimal bracketing principle says that paratactic structure should be flat:



Whereas immediate constituent grammars would form the list recursively, resulting in a tree:



Since the paratactic lists can be arbitrarily long, some recursive mechanism is needed to produce the structure. The solution suggested in the current systemic literature is the "recursive system," which amounts to a loop in the system network.



Hudson (1971, p. 61-62) claims:

> The big advantage of allowing recursion of this kind in the system network, rather than in rules that affect the structure directly, is that it does not add unwarranted structure. If we use a phrase-structure rule, such as 'x -> x+x' then we can generate nothing but binary structures, whereas we really want to generate single layers of structure with any number of ICs. In TG theory, the way has been found out of this dilemma by the introduction of a new bit of theoretical apparatus, the 'rule schema', but in systemic theory, no special apparatus is needed. We allow 'incomplete' to occur any number of times in the paradigmatic description of an item, by a recursive system, and then we map each occurrence of 'incomplete' onto a separate element in the item's structure.

Despite Hudson's claims, it is readily apparent that there are some problems with recursive systems. If features are the name of a class, then the distinction made here involves the number of times an item belongs to a particular class, which makes little sense. Hudson proposes to abolish the one-to-one correspondence between classes and features. This not only introduces new theoretical problems, but also does not really solve the original problem. For instance, feature x in the example is really only an entry condition for the first occurrence of the choice--not those following--

yet the entry condition in the network is always a disjunction involving x. More seriously, the functions in the realization rules inside the loop would need to be indexed (as Halliday and Martin, 1981, Glossary, say they are), and almost certainly require variables (e.g. (Fn ^ Fn+1)).

Given the problems with recursive systems, it is not surprising that they have not been implemented in previous text generation projects (e.g. Davey, 1978, says he is not convinced that recursive systems are the answer to the problem).

There are some general approaches that might lead to a satisfactory solution. All of these retain the one-to-one correspondence between features and classes. First, since it appears that even following Hudson's radical proposal the functions would have to be indexed, the result ends up looking very much like a schema anyway. Perhaps it would be best just to remove the recursive entry condition and treat features with indexed functions as schemas of some sort.

The suggestion in McCord (1975, p. 211) is that new realization relations could be introduced that operate uniformly on lists of structure nodes. For instance there could be a realization relation "listify" that is similar to "lexify" but associates a list of lexical items with a function. The problem here is that paratactic elements (for example) may not always be treated uniformly (e.g. "John, Bill, and Mary"). This suggestion has the advantage of not requiring an entirely new mechanism to be added to systemic grammar, but it is not clear that it will be sufficient in all cases.

There is no doubt that a replacement needs to be found for recursive systems. This is a major theoretical problem that severely restricts the abilities of systemic text generators. Nevertheless, no attempt has been made to remedy this situation here; system networks are required to be loop-free and parataxis has been avoided.

3.4. The strata

Halliday adopts

> ... the general perspective on the linguistic system you find in Hjelmslev, in the Prague school, with Firth in the London school, with Lamb, and to a certain extent with Pike--language as a basically tristratal system: semantics, grammar, phonology. (Halliday, 1978, p. 39)

It should be pointed out that "grammar" here refers to lexicogrammar, i.e. it includes vocabulary. Also, "phonology" should really be expanded to "phonology/orthography" to include writing as well as speaking (as Halliday often does elsewhere in his writings).

It is important to understand that the relationship between these strata is not one of delicacy; for instance, phonology is not just a more detailed continuation of the grammar. Each stratum has its own relationships and dimensions of abstraction--this is the point of stratification. Semantics, grammar and phonology are each described in terms most appropriate to that particular aspect of language. The result is three different but not independent representations of language.

Although the three strata are different representations, the representation language for the most part is the same. Each description is organized as systems of features. As stressed earlier, this means that the representation at each of the strata is a description of "potential."

3.4.1. The semantic stratum

The semantic stratum is a representation of the speaker's "meaning potential": using Halliday's gloss, this is what a speaker "can mean." For instance, suppose a mother wants to control the behaviour of her child by issuing a threat. There are two potential choices: she may threaten the child's privileges, or she may threaten some form of physical punishment. The semantic stratum will be discussed in detail in Sections 3.5 and 3.6.

3.4.2. The grammatical stratum

The grammatical stratum is a representation of what the speaker "can say" (in the sense of "formulate"). A typical choice here is between an indicative and an imperative clause. The grammatical stratum has already been discussed in some detail in this chapter.

3.4.3. The phonological/orthographic stratum

The phonological/orthographic stratum is a representation of how the speaker "can sound" or "can write." Typical kinds of choices here are whether or not to emphasize a particular word in the case of phonology, or what punctuation to use in orthography. This stratum will not be discussed further because it follows the same theoretical principles as the other two strata, and has not been implemented.

3.4.4. Interstratal preselection

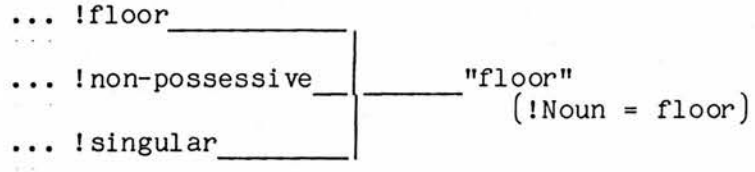
Although the strata have been presented as independent representations, they are clearly related. The relationships are represented through interstratal preselection, which is essentially the same as the preselection between ranks described earlier. Features at the semantic stratum may have realization rules which preselect grammatical features. Similarly, grammatical features may preselect features from the phonological/orthographic stratum. As Halliday (1973, p. 85) says:

> In general the options in a semantic network will be realized by selections of features in the grammar--rather than 'bypassing' the grammatical systems and finding direct expression as formal items.

Thus, sets of features at the semantic stratum are mapped, using preselection, onto sets of features at the grammatical stratum which are, in turn, mapped onto sets of features at the phonological/orthographic stratum. At the phonological/orthographic stratum there is no lower stratum from which to preselect, so the realization is in terms of physical characteristics instead of

features.

Note that there is no restriction on the rank from which grammatical features are preselected. The semantics can, and in most grammars probably must, preselect some features from each of the clause, group and word ranks. This means that the path of preselections from the semantics to the word rank can involve several steps. For instance, in the case of the Subject, the number is important at the clause rank, so the semantics may preselect the feature singular-subject. One of the realization rules of this feature preselects the feature singular from the nominal-group network. This in turn results in the feature !singular being preselected from the noun network (for the Head function). In other cases, for instance to preselect a lexical entry like !floor, the semantic stratum preselects the feature directly from the word rank. In other cases the semantic stratum chooses from the nominal-group network--for instance, to preselect features like non-possessive-nom, which preselects the feature !non-possessive for the Head. It makes no difference what the preselection path is. For instance all the above features at the word rank are entry conditions to a gate:



Since there is no phonological/orthographic stratum, the "lexify" realization rule above simply associates a lexical item with the function.

3.5. The semantic stratum

The semantic stratum, as it appears in the systemic theory, is particularly relevant to systemic work on text generation. This is because the semantic stratum must act as the interface between the extralinguistic inference and the grammar. Although systemic

grammar has been used in several text-generation projects, the semantic stratum as described in (Halliday, 1978) has never been included. For these reasons the semantic stratum will be given some extra attention here.

The term "semantic" in systemic theory has quite different connotations than it does in other branches of linguistics. In systemic linguistics "semantics" includes much of what is normally referred to as "pragmatics," and it is not represented or defined in terms of truth functions. Semantics here is directly related to Malinowski's notion of "context of situation." In fact Halliday originally used the term "contextual" to refer to this stratum.

> All language functions in contexts of situation, and is relatable to those contexts. The question is not what peculiarities of vocabulary, or grammar or pronunciation, can be directly accounted for by reference to the situation. It is which kinds of situational factor determine which kinds of selection in the linguistic system. (Halliday, 1978, p. 32)

Thus, in systemic theory, the context becomes the key to the semantics. Clearly, in this case a more precise notion of "context" is required. To this end Halliday and others have developed the idea of "register."

> Types of linguistic situation differ from one another, broadly speaking, in three respects: first, what is actually taking place; secondly, who is taking part; and thirdly, what part language is playing. These three variables, taken together determine the range within which meanings are selected and the forms which are used for their expressions. In other words, they determine the 'register'.
>
> The notion of register is at once very simple and very powerful. It refers to the fact that the language we speak or write varies according to the type of situation. This in itself is no more than stating the obvious. What the theory of register does is to attempt to uncover the general principles which govern this variation, so that we can begin to understand what situational factors determine what linguistic features. (ibid., pp. 31-32).

The three respects in which situations differ, as just described, are termed: field--"what is actually taking place";

tenor--"who is taking part"; and mode--"what part language is playing" (Halliday, 1978, passim). Field, tenor and mode are useful conceptual groupings that play a similar role to the metafunctions at the grammatical stratum. Also, like the metafunctions, they do not appear in the formal model.

3.5.1. Field

The field is the socially recognized physical setting in which text occurs, including the activities in progress.

3.5.2. Tenor

The tenor is a characterization of the relationship between the participants. This includes not just their respective social positions, discourse roles etc., but also the emotional issues of the moment.

3.5.3. Mode

Mode refers to the role language is playing in a particular situation. This involves characteristics of the text such as whether it is spoken harshly or written, and so on. It also involves the social function the text is performing, e.g. being descriptive, being persuasive etc.

Field, tenor and mode define the register of a social context. The semantic stratum is represented as a system network that specifies the choices available in field, tenor and mode--i.e. it is a paradigmatic description of register.

3.5.4. Register and metafunction

Halliday (ibid.) relates field, tenor and mode individually and as a group to the metafunctions at the grammatical stratum (see Section 3.3.8). Both register and metafunction provide broad organizational principles to explain the relationship between features or sets of features at their respective strata. Individually, field is

related to the ideational metafunction, by stating the general principle that semantic features associated with field tend to preselect ideational features. Tenor and the interpersonal metafunction have the same relationship, as do mode and the textual metafunction.

As an illustration, Halliday briefly describes two registers as follows (ibid., p. 226 and p. 115 respectively):

> Field: Instruction: the instruction of a novice
> - in a board game (e.g. Monopoly) with equipment present
> - for the purpose of enabling him to participate
>
> Tenor: Equal and intimate: three young adult males; acquainted
> - but with hierarchy in the situation (2 experts, 1 novice)
> - leading to superior-inferior role relationships
>
> Mode: Spoken: unrehearsed Didactic and explanatory, with undertone of non-seriousness
> - with feedback: question-and-answer, correction of error

> Field Child at play: manipulating movable objects (wheeled vehicles) with related fixtures, assisted by adult; concurrently associating (i) similar past events, (ii) similar absent objects; also evaluating objects in terms of each other and of processes.
>
> Tenor Small child and parent interacting: child determining course of action, (i) announcing his own intentions, (ii) controlling actions of parent; concurrently sharing and seeking corroboration of own experience with parent.
>
> Mode Spoken, alternately monologue and dialogue, task-oriented; pragmatic, (i) referring to processes and objects of situation, (ii) relating to and furthering child's own actions, (iii) demanding other objects; interposed with narrative and exploratory elements.

Here are some examples of interactions between register and the grammar: In the second example, when assistance from the adult is the subject matter, the ideational features related to benefaction are relevant to the field. Similarly, when similar events are recalled, the ideational feature past will be preselected. In the case of tenor, interaction with the parent will require preselecting

the interpersonal features concerning "person." Determination of course of events will mean preselecting interpersonal mood and polarity features. In the case of mode, reference to objects and situations will involve anaphoric and exophoric reference by preselecting textual features (ibid., p.117).

3.5.5. A closer look

Unfortunately, there has been little detailed work done on the semantic stratum, and several important issues are yet to be resolved.

For instance, one of the most important aspects of the grammatical stratum is the specification of grammatical structure. Halliday (1978, p. 41) admits:

> We know more or less what the nature of grammatical structure is. We know that constituent structure in some form or other is an adequate form of representation of the structures [at] the lexicogrammatical level. It is much less clear what is the nature of the structures [at] the semantic level.... [When working with the language of young children] it has been possible to bypass the level of semantic structure and go straight into lexicogrammatical constituent structure. That's all right for certain limited purposes. But there is obviously a limitation here, and when we attempt semantic representation for anything other than these highly restricted fields, it is almost certainly going to be necessary to build in some concept of semantic structure. But what it will look like exactly I don't know. I don't think we can tell yet.

For reasons of convenience, and since it seems to be adequate for the limited examples presented here, the realization rules at the grammatical stratum have been used at the semantic stratum as well. In other words, a simplifying assumption has been made that the structures at the semantic stratum are directly analogous to the structures at the grammatical stratum. This implies that there are semantic functions analogous to Agent, Subject etc.

3.6. Example

An example of the semantic stratum for a typical expert system domain would be ideal at this point, but unfortunately the only example from adult registers that Halliday presents in any detail is that of a mother threatening her child. Nevertheless, this will be sufficient to illustrate the ideas discussed in the previous section.

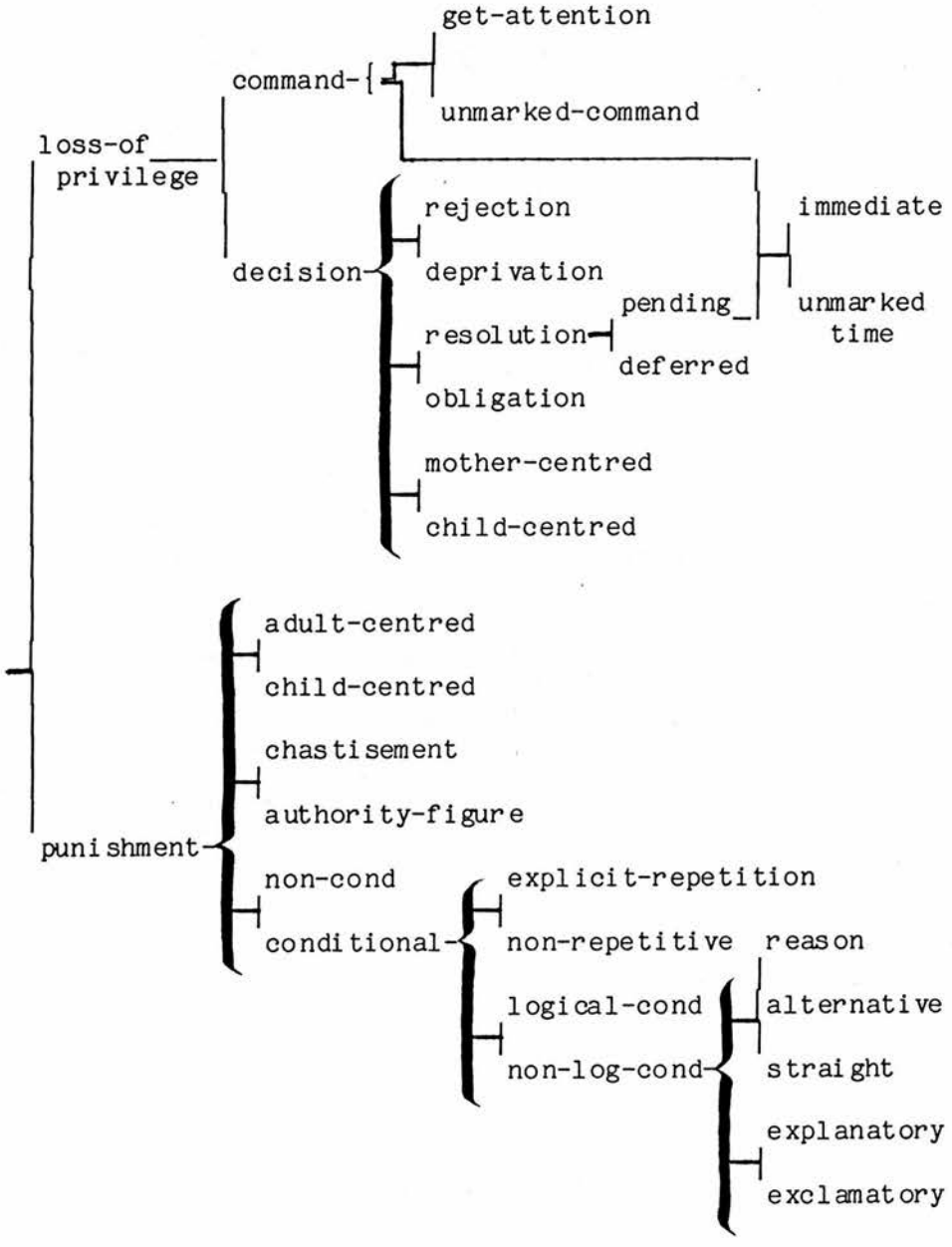


Figure 3.5
Some semantic choices

Figure 3.5 shows some systems from a semantic system network--the gates and realization rules have been omitted.

Note that this network represents only some of the choices for some very restricted registers. The choices illustrated here are mainly to do with tenor--the specific relationship between the mother and her child. For instance, in the case of explanatory on one hand, the mother is acting as an informant and backing it up with a threat; in the case of exclamatory on the other hand, the mother is setting herself up as the authority and telling what the consequences of disobedience will be. For instance, if the features straight, explicit-repetition, chastisement, adult-centred, non-log-cond, conditional, and punishment are chosen with explanatory, the result is "you mustn't do that, next time I'll smack you". If exclamatory is chosen instead, the text is "don't do that, next time I'll smack you". Alternatively, the mother may set herself up as an intermediary between the child and an "authority figure" who is to carry out the threatened action. For instance, suppose instead of chastisement above, the feature authority-figure is chosen. The resulting text may be "don't do that, next time Daddy will smack you".

When generating a clause, the choices for field and mode must also be made. The mode will almost certainly involve choosing a harsh tone of speech. The choices for field and mode may interact with the choices illustrated here for tenor. Field choices are likely to influence the choice of authority figure should the tenor require one. If the scene is set at home, the authority figure is very likely to be the father. If set elsewhere, the authority figure may be, for instance, a policeman (Halliday, 1978, p. 84). Of course the field may also influence the mode--the tone of voice may be lowered if there are other people present, and so on. Thus there may be a gate for a mode feature harsh-whisper that has entry conditions from both the field and tenor sections of the network.

It may appear that semantic features like people-nearby imply that the semantic stratum must represent every possible physical

situation. This is not the case--only those factors that are linguistically relevant for a particular speaker must be represented. The fact that there are people nearby may have a significant effect on the form the utterance should take; the fact that there is a person in the Empire State Building wearing red socks may not. The point is that there is a discernable set of factors which, for a particular speaker, are linguistically relevant during text generation. The semantic stratum represents these and their various interrelationships.

3.7. Summary

The purpose of this chapter has been to introduce and motivate those concepts from systemic linguistics that play a significant role in the rest of the thesis. The origin of many of these concepts can be traced back through Halliday, through Firth, to Malinowski. Malinowski's concept of an abstract "context of situation" was developed by Firth into the idea of "system" and developed further by Halliday into systemic grammar. Malinowski's emphasis on the cultural and social environment of language led to Firth's concept of the "multiplicity of languages" within a language and to Halliday's work on register and the semantic stratum. Malinowski's observation of the broad functions of language led to Halliday's idea of macro- and metafunctions--making systemic grammar a functional theory as opposed to just a syntactic theory with some functional labels attached.

Three goals of systemic grammar were then identified. They are: the description of the function of language, the description of the structure of language, and the classification of linguistic alternatives.

Some specific concepts from systemic grammar were then discussed, and illustrated at the grammatical stratum. It was then pointed out that there are in fact three strata: the semantic, the grammatical, and the phonological/orthographic. Since the preceding discussions were primarily concerned with the grammatical stratum,

and since the semantic stratum plays a particularly important role in the approach to text generation described later, the semantic stratum was then discussed in some detail.

> ... slang is ... often used by people who are deliberately adopting a certain speech variant for social purposes. (Halliday, 1978, p. 158)

4. The Conflation

The previous two chapters have discussed the independent fields of AI problem solving and systemic grammar. This chapter will point out that in fact there is an important relationship between the two fields that can form the basis for a "Systemic Linguistic Approach to Natural-language Generation" (SLANG). The first few sections will describe the various facets of the relationship between AI problem solving and systemic grammar, and the text-generation method that results. Then some examples will be presented to illustrate the text-generation method just described. Finally there is a short discussion of the significance of this approach to text generation.

4.1. The fundamental relationship

> The central nature of intelligent problem solving is that a system must construct its solution selectively and efficiently from a space of alternatives. (Hayes-Roth et al., 1983a, p. 20)

> We shall define language as 'meaning potential': that is, as sets of options or alternatives, in meaning, that are available to the speaker-hearer. (Halliday in deJoia and Stenton, 1980, #572)

Compare these two quotations. The fields of study examined in the previous two chapters are both organized around a space of alternatives. Notice that these passages do not refer to peripheral issues; the first few words of each, "The central nature of intelligent problem solving is ..." and "We shall define language as ...," indicate that the issues involving alternatives lie at the nucleus of the respective disciplines. This being the case, there is clearly a fundamental relationship between AI problem solving and systemic grammar. This section will probe into this fundamental relationship, in an attempt to discover its origins and nature.

4.1.1. Alternatives in AI problem solving

AI problem solving is characterized as a "search" through a space of alternatives. Chapter 2 discussed some of the techniques employed by AI problem solvers over the years to find a solution within a space of alternatives. The techniques ranged from blindly searching through the possibilities until a solution was found, to efficient goal-directed knowledge-based techniques that selectively considered only alternatives that may lead to a solution. Whether the alternatives are explicitly searched or whether they are avoided, the entire space of alternatives is always at least implicitly represented.

4.1.2. Alternatives in systemic linguistics

The emphasis on alternatives in systemic linguistics originated in two separate aspects of Malinowski's work (e.g. 1923). He characterized language as an action, an integral part of the everyday actions in a society. He also argued that language can only be understood in a specific context. The second point was developed later by Firth when he precisely stated, and indeed defined, the context of language in terms of "potential":

> ... Firth built his linguistic theory around the original and fundamental concept of the 'system', as used by him in a technical sense; and this is precisely a means of describing the potential, and of relating the actual to it....
>
> The potential of language is a meaning potential. This meaning potential is the linguistic realization of the behaviour potential; 'can mean' is 'can do' when translated into language. The meaning potential is in turn realized in the language system as lexico-grammatical potential, which is what the speaker 'can say'. (Halliday, 1973, pp. 50-51)

Thus in systemic linguistics, the starting point is the set of alternatives in meaning. Linguistic contexts are characterized by the alternatives in meaning available in the particular context--meaning potential. These alternatives are realized by, or mapped onto, sets of grammatical alternatives.

One important point also apparent from the quotation is that systemic theory treats extralinguistic matters in terms of potential as well. An agent in a particular (social rather than linguistic) context has a "behaviour potential"--what the agent "can do." Some of the alternatives in social situations are linguistic, and these form the meaning potential. The crucial point is that the linguistic alternatives are just a subset of the behavioural alternatives that can realize the behavioural potential.

Now it must be understood that the notion of alternatives--paradigmatic description--plays a more central role in systemic grammar than in other linguistic theories:

> If we go back to the Hjelmslevian (originally Saussurean) distinction of paradigmatic and syntagmatic, most of modern linguistic theory has given priority to the syntagmatic form of organization. Structure means (abstract) constituency, which is a syntagmatic concept. Lamb treats the two axes together: for him a linguistic stratum is a network embodying both syntagmatic and paradigmatic relations all mixed up together, in patterns of what he calls AND and OR nodes. I take out the paradigmatic relations (Firth's system) and give priority to these; for me the underlying organization at each level is paradigmatic. (Halliday, 1978, p. 40)

In Halliday's theory, the alternatives are dependent on other alternatives, not on structures.

> ... and here I depart from Firth, for whom the environment of a system was a place in structure--the entry condition was syntagmatic, whereas mine is again paradigmatic. (ibid., p. 41)

The important point to note here is that while many grammatical theories have alternatives as a (perhaps even an important) consideration, they are usually first and foremost theories of structure, and the representation of alternatives is sacrificed to this end. Whereas in systemic grammar the alternatives are primary.

> By 'text', then, we understand a continuous process of semantic choice. Text is meaning and meaning is choice, an ongoing current of selections each in its paradigmatic environment of what might have been meant (but was not). It is the paradigmatic environment--the innumerable

> subsystems that make up the semantic system--that must provide the basis of the description, if the text is to be related to higher orders of meaning, whether social, literary, or of some other semiotic universe. (ibid., p.137)

Thus in systemic linguistics, paradigmatic description--description in terms of alternatives--is the crucial representational concern.

4.1.3. The fountainhead

Noting the fact that AI problem solving and systemic grammar are both organized around alternatives is only the first step. Next it must be noted that in knowledge-based AI problem solving, the alternatives represent the problem--knowledge about the alternatives is then required to guide the problem solver to a solution. Systemic grammar is knowledge about linguistic alternatives; the entry condition and realization rules specify the conditions and effects of a particular alternative--exactly the information required by an AI problem solver. Thus the primum mobile of this work becomes apparent: a systemic grammar can be interpreted as linguistic problem-solving knowledge and used by an AI problem solver to find--selectively and efficiently--the solution to linguistic problems in exactly the same way as knowledge from other domains is used to solve problems in those domains.

4.2. The conflation

A particularly important consequence of the fundamental relationship between AI problem solving and systemic grammar is that the central representations found in each of the two fields is equivalent. This means that a systemic grammar can be directly interpreted as both linguistic description and problem-solving knowledge simultaneously--i.e. the two interpretations can be conflated. This conflation provides the impetus for a new approach to text generation, but is only the beginning. The conflation reaches much further than just the surface representation; it extends to the foundations of systemic theory.

The discussion here, and throughout the remainder of the thesis, will involve describing the relationship between systemic grammar and one particular variation of the AI representation: production rules. A production system is only one of several architectures able to selectively and efficiently process a space of alternatives. Production rules were chosen here for reasons of simplicity, accessibility and for their formal properties (see Chapter 6). It is important that the reader understand that similar expositions could be given for representations such as "objects" (see Section 7.5).

A brief comment should be made here about the role of the system networks in the SLANG model. Halliday is careful to state that the system networks merely describe the "meaning potential" of language, and careful not to state or imply that the system networks themselves play any role whatsoever in the "actualization" of that meaning potential.

> ... when we examine the meaning potential of language itself, we find that the vast numbers of options embodied in it combine into a very few relatively independent 'networks'; and these networks of options correspond to certain basic functions of language. This enables us to give an account of the different functions of language that is relevant to the general understanding of linguistic structure rather than to any particular psychological or sociological investigation. (Halliday in deJoia and Stenton, 1980, #541)

The system networks in SLANG <u>are</u> directly involved in the actualization of the meaning potential--they are exploited as problem-solving knowledge. While this is not a contradiction of Halliday's position, it is nevertheless a different interpretation than is normally given to system networks. It is worth reiterating that the term "knowledge" here is used only in the AI sense; no claims are being made about the structure of human knowledge (see Halliday, 1978, pp. 38, 51).

4.2.1. Conflating representations

Since both problem-solving knowledge and systemic grammar must describe the complex relationships between interdependent alternatives, it is not too surprising that they developed the same basic representations. For each alternative, the conditions under which the alternative is applicable must be represented, as must the effects or consequences of the particular alternative. Both AI problem solving and systemic grammar have adopted this two-part representation.

4.2.1.1. Conflating gates and forward-chaining rules

Notice that gates, as described in Chapter 3, are represented in terms of conditions and effects. Gates have a set of entry conditions and a set of realization rules. Consider the gate feature does represented here in systemic notation:

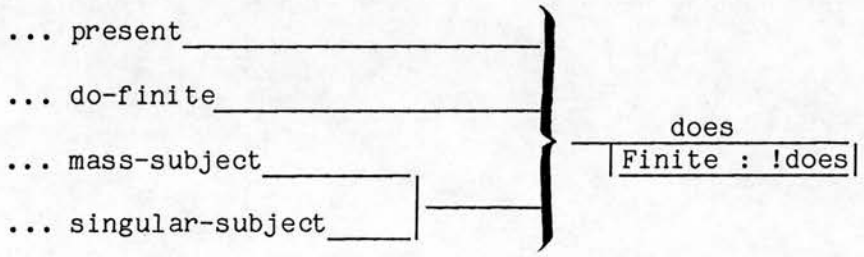


Figure 4.1
A gate (Mann/Halliday).

If the features present and do-finite have been selected, and either

mass-subject or singular-subject has been selected, then the feature does must be selected. The effect or consequence of choosing this feature is that the lexical feature !does is preselected, so the Finite element will be realized by one of "does", "doesn't" or "does not."

Now, since production rules, an AI problem-solving representation, also have conditions and effects, gates can be interpreted as production rules. E.g.:

> if present and do-finite and
> one of singular-subject or mass-subject
> have been chosen,
>
> then choose does and preselect the lexical
> feature !does for the Finite.

This can be used for simple forward-chaining as described in Chapter 2. Interpreting entry conditions of a gate as the LHS of a production rule, and the choice and realizations as the RHS, corresponds to the intuitive interpretation of a gate: if the logical combination of features acting as the entry condition to a gate feature is satisfied, then choose the feature and constrain/modify the structure of the text according to the realization rules.

Thus we can interpret a gate in any of these representations (system network, or the various production notations, e.g. see Chapter 7 for the OPS5 representation) either as a piece of a systemic grammar, or as a piece of problem-solving knowledge. In fact it is advantageous to conflate these interpretations--make both interpretations simultaneously.

4.2.1.2. Conflating systems and backward-chaining rules

Suppose the features in a system network that are terms in a system are interpreted similarly. For every feature in a system there will be one rule stating:

if the entry condition of the system is satisfied,

then choose this feature and perform the actions specified by the realization rules.

Notice that if these rules are interpreted as forward-chaining rules, they are not much use to the problem solver, since it doesn't know which of the alternatives should be chosen. Specifying in the representation that the terms in a system are mutually exclusive doesn't help. The technique of backward-chaining will be used instead. So the interpretation of the rule above is:

if there is a goal to choose this feature, or a goal that can be satisfied by one of the realization rules,

then choose this feature and set the entry conditions as subgoals.

E.g.

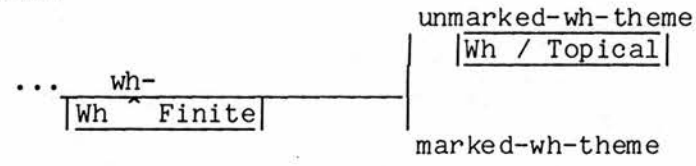


Figure 4.2.
A system (Mann/Halliday).

The feature unmarked-wh-theme can be interpreted as a backward-chaining rule:

if there is a goal to choose unmarked-wh-theme
or there is a goal to conflate the Topical with the Wh element,

then choose unmarked-wh-theme
and set a subgoal to choose wh-.

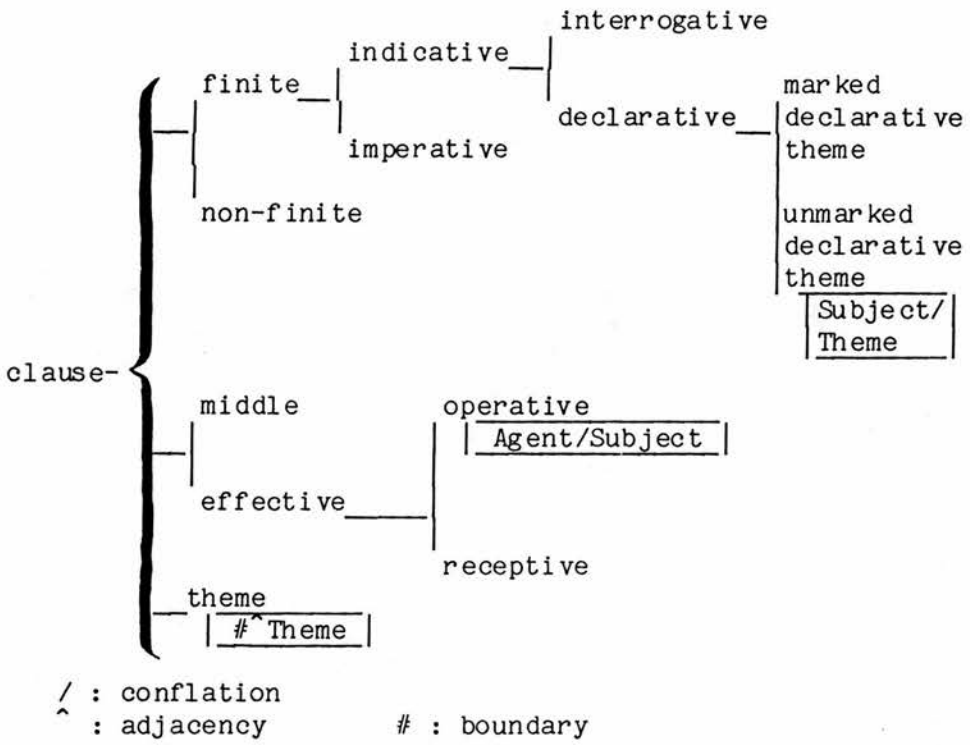
Again this rule could be written in any of the various production notations as well as the system network notation.

4.2.1.3. Conflating the grammar and the knowledge base

Thus all the features in a systemic grammar, together with their entry conditions and realization rules, whether they form gates or systems, can be interpreted as problem-solving rules of the kind used by AI problem solvers. This means that the grammatical stratum as a whole can be interpreted as a knowledge base (more likely part of a larger knowledge base) of grammatical knowledge. This knowledge can be used to solve grammatical problems in exactly the same way as medical knowledge can be used to solve medical problems, and chemistry knowledge can be used to solve chemistry problems.

4.2.2. Conflating text generation with problem solving

Having shown that the fundamental relationship between systemic grammar and AI problem solving allows the systemic representation to be interpreted both as a grammar and as problem-solving knowledge, it will now be possible to show that the process of systemic text generation can be conflated with the process of problem solving.



/ : conflation
^ : adjacency # : boundary

Figure 4.3. A grammar excerpt.

Consider the simplified systemic fragment in Figure 4.3, and imagine a hypothetical problem solver that can perform forward- and backward-chaining. Suppose the semantics sets the goal to conflate the Agent and the Theme. This is a grammatical problem that can be solved using the grammatical knowledge contained in the grammar. This goal cannot be solved immediately since no feature has a realization rule conflating these two functions. However, assuming there is a general rule expressing the transitivity of conflation, this rule can set as subgoals: the conflation of the Agent with X, and the conflation of the X with the Theme--where X can be instantiated to Subject. The features operative and unmarked-declarative-theme respectively have these realization rules (effects) so the backward-chaining begins there.

The feature operative, to start with, has the entry condition effective. So effective becomes a subgoal. It in turn has the entry condition clause which then becomes a subgoal. This chain of reasoning stops once clause is chosen because it has no entry

conditions. Similarly, the problem solver will backward-chain from unmarked-declarative-theme through declarative, indicative, finite and clause. This backward-chaining is a similar idea to "path augmentation" described in (Mann et al., 1983, p. 68) for inter-rank preselection.

Sometime after clause is chosen in the example, the gate theme--interpreted as a forward-chaining rule--will fire, since its entry condition is satisfied. Although this is the only gate in this example, there will be many gates firing like this in a large grammar. The gates may fire in chains because many gates have other gates as their entry conditions.

E.g.:

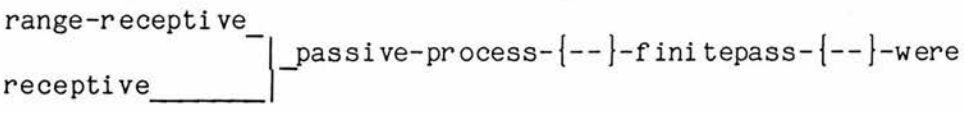


Figure 4.4
An excerpt from (Mann/Halliday)

Figure 4.4 shows a series of gates. If either range-receptive or receptive (neither of which is a gate) is chosen, passive-process fires. One of the gates for which passive process is one of the entry conditions is finitepass. One of the gates for which finitepass is one of the entry conditions is the gate were.*

It is important to understand that features such as declarative and finite in the original example, and finitepass and were in the gate excerpt, have realization rules (not shown here) that become side effects of the solution to the original goal.

Returning to the original example, there must be several goals of this kind set by the semantics, and after all the forward and backward reasoning has been done, and all the realization rules have

---

* Note that there is no choice here, so the input simply propagates through these gate networks like a logic circuit--presumably this is the origin of the term "gate."

been processed, the linguistic element will be uniquely determined. A small amount of additional domain-specific (i.e. linguistic) knowledge about conflation, adjacency and so on will enable the problem solver to actually construct the clause.

The crux of the matter is that there is no special mechanism here. The problem solver is using grammatical knowledge in exactly the same way as it can use other knowledge in other domains. Thus the process of text generation as described in this section has been conflated with the process of problem solving.

4.2.3. Conflating the semantic stratum with compiled knowledge.

The problem-solving process described in the last section would work. There may, however, be several different ways to achieve any particular goal, some of which may lead to conflicts with other goals and thus to backtracking. This method would have the advantage of being simple; but the disadvantage, of course, is that the backtracking makes even the most common and simple semantic goals very expensive to achieve. AI problem solvers avoid having to solve the same difficult problem repeatedly by "compiling" the result.

It is not a new idea that language involves difficult problems that occur repeatedly and thus is a good candidate for knowledge-compilation techniques. McDonald (1983a, p. 265) says his system cannot do

> ...planning by backwards chaining from desired effects....the effects of such instructions can sometimes be achieved "off-line" however, by having the designer precompute the decision-space that the deliberation would entail and then incorporate it into the component's library as what would in effect be an extension to the rules of the grammar.

Also, as Berwick (Brady and Berwick, 1983, p. 26) says:

> ... it seems hardly likely that every time one hears "Can you pass the salt?" one runs through in toto a long chain of inferences that ends with the conclusion that what was really meant was that someone wants you to pass the salt. The obvious alternative is to squirrel away some commonly occurring deductions ... Of course this approach begs an

important research question about the nature and organization of these ...

Berwick's comments clearly apply to generation as well, and the answer to the begged question is Halliday's semantic stratum (as described in Section 3.5). The nature of the compiled plans or deductions is that they associate grammatical features with situations; their organization is by register. Thus the semantic stratum can be conflated with the high-level compiled knowledge found in AI problem solvers.

Of course the solutions can be compiled to various degrees. For instance any semantic feature that preselects the grammatical feature unmarked-declarative-theme in the sample grammar above could also preselect the features declarative, indicative, finite and clause--thus compiling part of the backward-chaining process described earlier. There appears to be a tradeoff between clarity and conciseness on one hand, and speed on the other. The approach taken in this work is to have the semantic features preselect only enough grammatical features to determine the result. There are several reasons for this decision. First, in research of this nature clarity is essential. Second, the backward-chaining process is very efficient and does not introduce a large overhead. Third, in the deeply compiled version, making a small change to the grammatical stratum would require many changes to the semantic stratum (unless the compilation is automatic--see 9.3.3).

The question is now: exactly which features need to be preselected to determine the result? The answer lies in the various sources of disjunction in the system network. The major sources of disjunction are the systems. Since features that are terms in systems are interpreted as backward-chaining rules, many of these features will be chosen if and only if they are part of the entry condition of a more delicate feature.

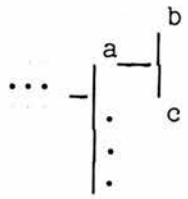


Figure 4.5

Consider Figure 4.5. There is simply no point in preselecting the feature a, because then either b or c must be chosen somehow, and whatever mechanism chooses between them (backward-chaining or preselection) will thereby imply a, making the original preselection redundant. Thus it becomes clear that there is no point in preselecting a feature in a system if it is part of the entry condition to another system.

Hudson (1981, p. 214) makes this point when discussing the interface between the grammatical and the phonological strata:

> [T]he only phoneme features that we need to specify as realization for a lexical item are the ones on the right-hand edge of the system-network. This obviously constitutes a major economy in the rules.

The exception to this rule is a result of another form of disjunction: disjunctive entry conditions to systems.

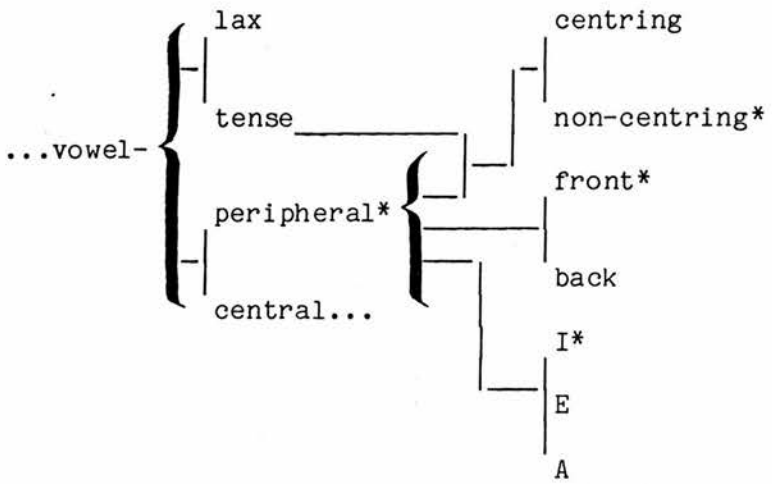


Figure 4.6
A fragment from a phonological network (Hudson, 1981, p. 213)

Consider Figure 4.6 (The asterisks indicate unmarkedness and will be referred to in Section 9.3.2.3). The features "on the right-hand edge of the system network" i.e. those features that do not have systems to their right are: lax, centring, non-centring, front, back, I, E and A. Suppose non-centring is preselected. At this point the problem solver does not know which of the two disjunctive entry conditions (tense, peripheral or both) to set as subgoals, so it cannot continue backward-chaining. On one hand, if the vowel is in fact peripheral, then a feature must be preselected from each of the other two systems (front/back and I/E/A). In this case peripheral will be chosen during the course of backward-chaining from each of these two systems and this gratuitously solves part of the problem with the disjunctive entry condition to non-centring. On the other hand, if the vowel is in fact tense there is no way to infer this by backward-chaining from other systems, so it must be preselected.

Disjuncts like peripheral that are dependent on other systems and not disjunctive with respect to those, are termed dependent disjuncts. Disjuncts like tense that have no such dependents and thus cannot be resolved by backward chaining are termed independent disjuncts. The rule then, is that features that do not have systems to their right or are independent disjuncts (collectively termed seed features) need to be preselected, and all other features can be deduced from these (this will be proven in Chapter 6).

Note that gates never need to be preselected. Even if gates have disjunctive entry conditions, it makes no difference because the chaining is in the other direction. Recall Figure 4.1: mass-subject and singular-subject are disjuncts, but it does not matter which one has been chosen; the rule will still fire.

There is only one difficult case: features that are not terms in systems themselves but are part of the entry conditions to systems. In fact this happens in the (Mann/Halliday) clause network.

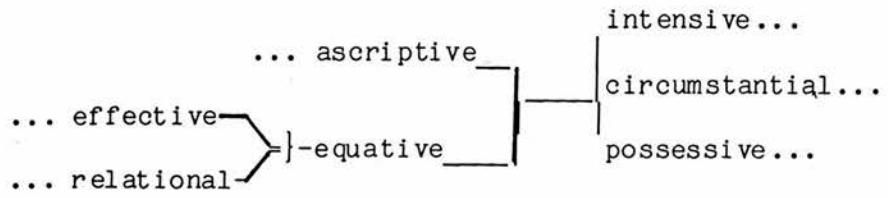


Figure 4.7. A fragment from (Mann/Halliday)

In this fragment equative looks like a gate, but in fact must be interpreted as a backward-chaining rule since it may act as the entry condition to a system. Notice also that it (like ascriptive) is an independent disjunct. Therefore the semantics must preselect either ascriptive or equative if the backward-chaining will run into this disjunction.

The semantic stratum thus acts as a layer of highly-compiled knowledge that guides the problem solving at the grammatical stratum by preselecting seed features (features that do not have systems to their right or are independent disjuncts). This is illustrated in Section 4.3 below.

Another topic is the possibility of the problem solver using the grammatical knowledge directly (as in the previous section) in cases where no appropriate compiled/semantic knowledge exists.

> As new, unanticipated patterns crop up, inflexible, compiled solutions fail. General problem-solving abilities allow a more graceful degradation at the outer edges of domain knowledge. (Brachman et al., 1983, p. 46)

It is possible to envisage a text-generation system that when "unanticipated registers crop up" could reason "from first principles" using the knowledge at the grammatical stratum. This will not be taken further here--see section 9.3.4.

4.2.4. Conflating behaviour potential and general problem-solving knowledge

It has already been mentioned briefly that systemic theory views "can mean" as one form of "can do"; meaning potential is one

form of behavioural potential. This nicely completes the correspondence between AI problem solving and systemic theory, since even the non-linguistic aspects of a problem-solving system can be related to the theory. Consider, for instance, a planning system for the blocks world. In a particular situation the planner may have several rules indicating valid actions that can be performed. It makes perfect sense to interpret this as the system's "behaviour potential." Perhaps one of these actions is a linguistic request to another agent to move a block. In this case the behavioural potential is also a meaning potential. The ability to relate linguistic issues to the larger behavioural sphere of activity will prove useful in cases where non-linguistic modes of communication are possible (see Appelt, 1982, 1983, and Sections 8.2.1, 8.3.1).

4.3. An example

Ideally, an example from a typical expert-system application would be given here, but unfortunately the only semantic stratum available is a fragment of a network for a mother threatening her child (described briefly in Section 3.6). The following example is meant to serve as an analogy to text generation in AI applications.

Suppose there exists a situation involving two agents: a mother and her child. The child has performed some action and the mother, in order to achieve some parental or other social goal, plans to prevent the child from repeating the act. As a result of some reasoning which is not at issue here, the mother decides that solutions such as physically restraining the child and so on would conflict with other goals. Another alternative, however, is to achieve this goal verbally. The task thus becomes a "text planning" task. The mechanism that performs this task is referred to as the "text planner" although it may well be the same general-purpose problem solver working with linguistic knowledge, not necessarily a special-purpose text-planning mechanism.

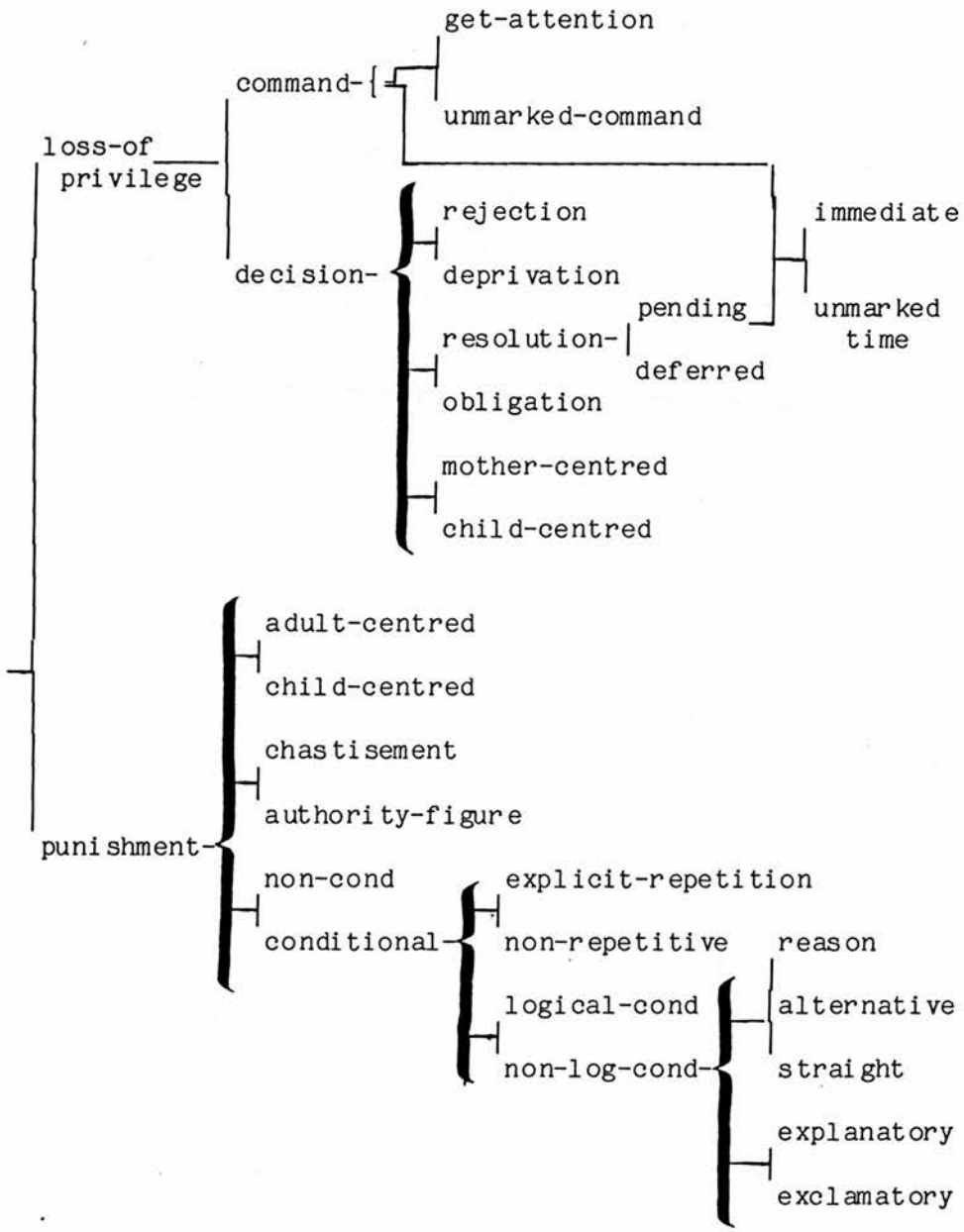
Again as a result of reasoning which is not at issue here, the text planner decides it can achieve the goal with a two-part

utterance: first chastising the child, explaining that the action should not be performed; second threatening punishment if the action is repeated. The text planner reduces the goal of creating this utterance to a set of semantic goals. It is now the job of the text generator to take these semantic goals and produce the natural-language output.

It will be assumed that the text generation is performed using the SLANG approach--a general purpose problem solver uses the knowledge contained in the grammar (semantic and grammatical strata) and some knowledge about systemic realization to generate the text.

To avoid confusion, semantic features are prefixed by "$" to help the reader easily distinguish them from grammatical features (there is no linguistic or computational significance placed on the prefix). Also, where there may be doubt, the network containing a feature will follow the feature in parentheses--e.g. declarative (clause).

The semantic goals set by the text planner are: $straight-threat, $explicit-repetition, $chastisement, $smack, $explanatory-cond and $adult-centred-punishment. These semantic goals are seed features of the following excerpt from a semantic stratum to which the text generator has access.



Some semantic choices

The features corresponding to these goals will be chosen and their entry conditions will be set as subgoals. The problem solver backward-chains from $explanatory-cond to $non-logical, from $explicit-repetition to $conditional, from $adult-centred-punishment to $punishment, from $chastisement to $punishment, from $non-logical to $conditional, from $conditional to $punishment, and from $punishment to $threat. Notice that several backward chains often pass through the same goal. The problem solver only achieves the goal

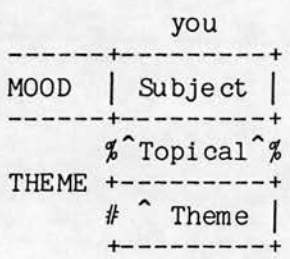
once, and if it is set again, it is recognized as being redundant. Sometime during the backward-chaining, some forward-chaining rules (not shown in the above figure, but listed in Appendix C, Section 13) also fire, viz. choosing $explanatory-cond triggers $stated-cond, and choosing $explicit-repetition and $straight-threat triggers $repeat-straight, while choosing $adult-centred-punishment and $chastisement triggers $mother-punishes.

The realization rules of these semantic features (See Appendix C, Section 13) preselect seed features from the grammatical stratum--they set grammatical goals. In fact the problem solver may not wait until all the semantic goals are solved before it begins to solve for goals at the grammatical stratum. The generation will be described word by word from left to right, starting at the top for each word in turn. This is not necessarily the exact order in which the problem solver attacks the goals but in fact is similar to the implemented problem-solving process described in Chapter 7. Many of the details, especially at the grammatical stratum, will be glossed over to make the description comprehensible. Nevertheless, it is hoped that the general idea will be conveyed.

The feature $conditional divides the text up into two parts: a condition and a threat, represented by the semantic functions $Cond and $Threat. There is also an adjacency realization rule ($Cond ^ $Threat) that orders the condition before the threat. This semantic adjacency is not the proper way to do this, but has been used as a shortcut. There should be a clause-complex rank at the grammatical stratum that handles this kind of clause ordering (Halliday, 1985, Chapter 7). So first the generation of the condition will be described.

The feature $stated-cond has realization rules which preselect, or set as goals, the features unmarked-declarative-theme and non-attitudinal (both clause features). The former has the effect of conflating the Topical and the Subject. The feature $non-logical-cond, which was inferred from $explanatory-cond or $straight-threat, sets the goal non-textual-theme (clause). When non-textual-theme

and non-attitudinal are chosen, they trigger a forward-chaining rule that together with other inferences orders the Topical at the front of the $Cond clause (by ordering Theme as the leftmost function in the clause, and Topical as the leftmost subfunction of Theme). Since the Topical is conflated with the Subject, this also means that the Subject is at the front of the clause. The feature $conditional also sets the grammatical goal addressee-subject (clause) which in turn has the effect (Subject = you)* and sets the goal !second-person (verb) for the Finite. Since the Subject is completely realized, and since it is the first item in the clause, it can now be output. The following is the current structure of the first clause:



The feature $explanatory-cond sets the goal modal (clause), and sets the goal !must (verb) for the Modal element.** The feature modal, chosen to achieve the above goal, has the effect (Modal /

---

* This could also be done by preselecting the appropriate features from the pronoun part of the noun network. Since the lexical item is in fact completely determined, lexification has been used instead.

** In fact, in this case "must" is not a modal, but rather what Halliday (1976b) calls a "quasi-modal" [!] or modulation (see also Halliday, 1985, p. 86). This is the difference between "Mary can't think that!" and "Mary can't think period!" (Halliday, 1976b). The former uses a modal to indicate that what is being said is obvious (the modal plays an interpersonal role); the latter uses a quasi-modal to indicate Mary's inability (the modulation plays an ideational role). Since the grammar used for the implementation cannot handle modulation, and since the syntax for the two is often identical, the current system pretends quasi-modals are true modals. This cheat prevents the implementation of the $obligation examples such as "you'll have to go upstairs" since, unlike modulations, modals can never be combined with the future tense.

Finite). At some point the problem solver backward-chains from unmarked-declarative-theme to infer the feature declarative, which has the effects (% ^ Subject), (Finite ^ %) and (Subject ^ Finite). This, together with the fact that Modal and Finite have been conflated, means that the Modal is the next item to be generated. Another of the effects of $non-logical-cond is that it sets the goal unmarked-negative (clause). At some point the problem solver infers indicative and finite from declarative by backward-chaining. These have the realization rules (Mood(Subject)) and (Residue ^ #), and (Mood(Finite)) and (Mood ^ Residue) respectively. From indicative and unmarked-negative the feature reduced-negfinite is inferred by forward-chaining, and sets the goal !reduced (verb) for the Finite. The feature negative is a condition for unmarked-negative and is thus inferred by backward-chaining. This, and the feature indicative, result in negative-finite being inferred by forward-chaining. The latter feature sets the goal !negative for the Finite. When the goals !negative, !reduced, and !must (all verb) are satisfied by choosing the corresponding features, "mustn't" is inferred by the forward-chaining rule shown in Figure 4.8.

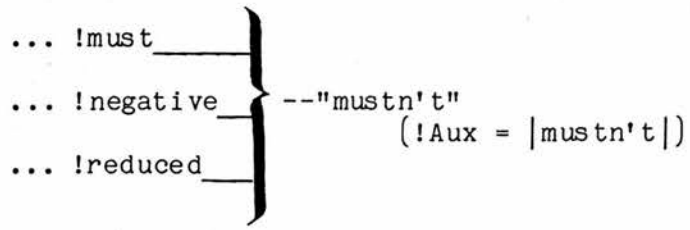
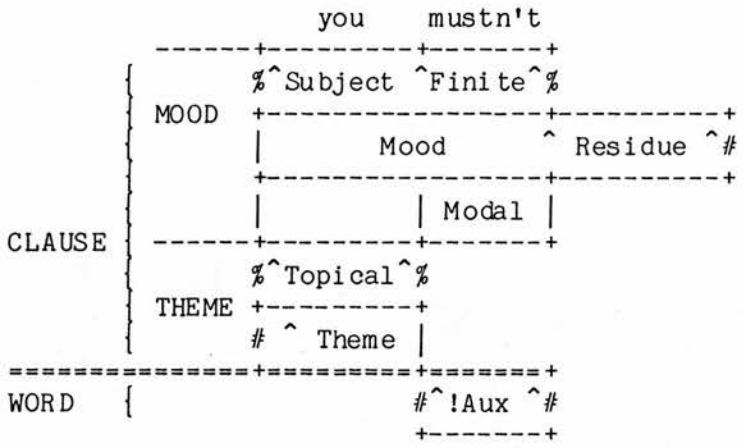
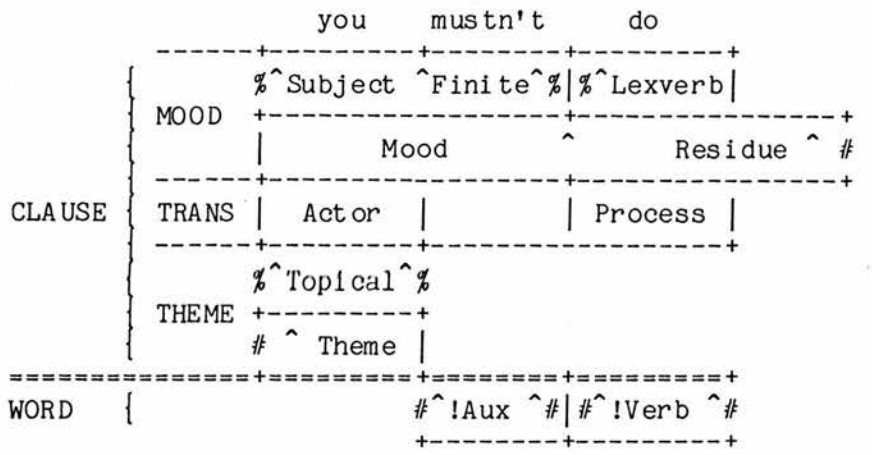


Figure 4.8

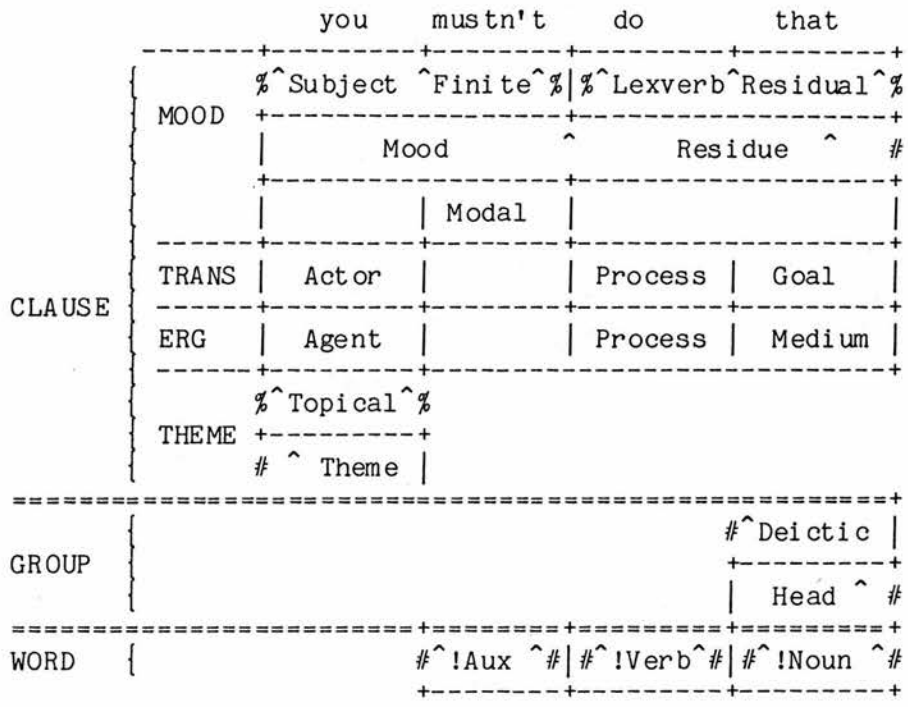
The lexical item "mustn't" thus realizes the Finite/Modal and can now be output.



The realization of the next item to be output, the Process, is again started by the feature $conditional, which sets the goal !-do- for the Process. This together with !stem triggers the feature "-do-" which lexifies the verb as "do" (the hyphens distinguish the lexical verb "do" from the auxiliary "do", but both are realized by the same item in the end). The feature !stem is inferred as follows: modal was an effect of $explanatory-cond; non-past-in (non-perfective) and non-present-in (non-progressive) were both effects of $conditional. From these, together with active-process, inferred from operative by forward-chaining, can be inferred modalstemlexverb--which has the effect of setting the goal !stem for the Lexverb. However this is also a goal for the Process since the Lexverb and the Process are conflated as an effect of clause, which will be inferred at the end of some backward-chaining process. The current relevant structure is:

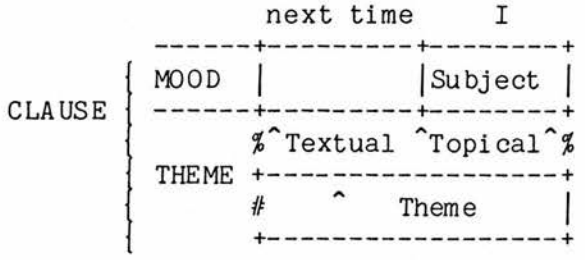


Finally, the grammatical function Goal is generated starting from a number of preselections from the nominal-group network by the semantic feature $conditional. It sets the following nominal-group goals: non-possessive-nom, singular, determiner-head and far. The two relevant functions in the nominal-group are the Deictic and the Head, which are conflated in this case by determiner-head. The feature far has the conditions demonstrative and non-interrogative-det which are thus inferred by backward-chaining. These, together with singular, trigger a forward-chaining rule that sets the goal !singular-pronoun for the Deictic. From !singular-pronoun and !singular (set as a goal by singular) the feature "that" is inferred, which sets the last lexical item in this clause to "that." Thus the conditional clause "you mustn't do that" has been generated. The final structure is:



Next, the threat clause must be generated. The semantic feature $non-logical-cond sets the goal textual-theme, which means that the Theme of this clause will consist of both a Textual and a Topical. The feature $repeat-straight sets the goal thesis-repetitive (conjunction), which has the effect of lexifying the function Time as "next time". Backward-chaining from there results in Time being conflated with Conjunct, which is then conflated with Textual (an effect of textual-theme (clause)). Since the Textual element is ordered first (another effect of textual-theme), "next time" can be output.

The feature unmarked-declarative-theme, set as a goal by $punishment, again conflates the Subject with the Topical, the next item to be generated. The semantic feature $mother-punishes sets the goal speaker-subject (clause), which has the effect (Subject = I), lexifying the Subject. "I" can now be output.



The features future and unmarked-positive (both clause) are set as goals by $punishment. The feature future sets the goal !will (verb) for the Finite. The features declarative and indicative are inferred by backward-chaining from unmarked-declarative-theme. The feature interactant-subject is inferred by backward-chaining from speaker-subject. From declarative, unmarked-positive, and interactant-subject, the feature reduced-posfinite is inferred by forward-chaining, which sets the goal !reduced (verb) for the Finite. The feature positive (clause) is inferred from unmarked-positive, and this, together with indicative, allows the problem solver to infer positive-finite, which sets the goal !positive (verb) for the Finite. The forward-chaining rule shown in Figure 4.9 then fires, realizing the Finite as "'ll".

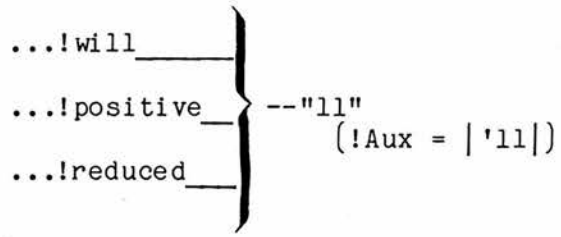


Figure 4.9

Since declarative orders the Finite just after the Subject, the Finite can be output.

The semantic feature $smack sets the goal !smack for the Process, the next item to be generated. Just like in the last clause, the feature modalstemlexverb is chosen, setting the goal !stem (verb) for the Lexverb, which is again conflated with the Process. The result is that the Process is realized as "smack".

Finally, the semantic feature $adult-centred-punishment has preselected the nominal-group features non-possessive-nom, personal, and singular for the Goal of the $Threat, and the word-rank features !second and !objective for the Head of the Goal of the $Threat ($Threat<Goal<Head). The nominal-group feature personal implies the feature pronoun, which together with singular triggers a forward-chaining rule that has the effect of setting the goal !singular-pronoun for the Head. The forward-chaining rule shown in Figure 4.10 then fires, realizing the Head (the only part of the Goal) as "you".

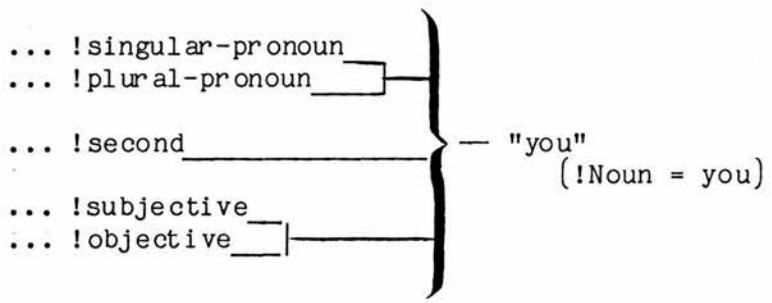
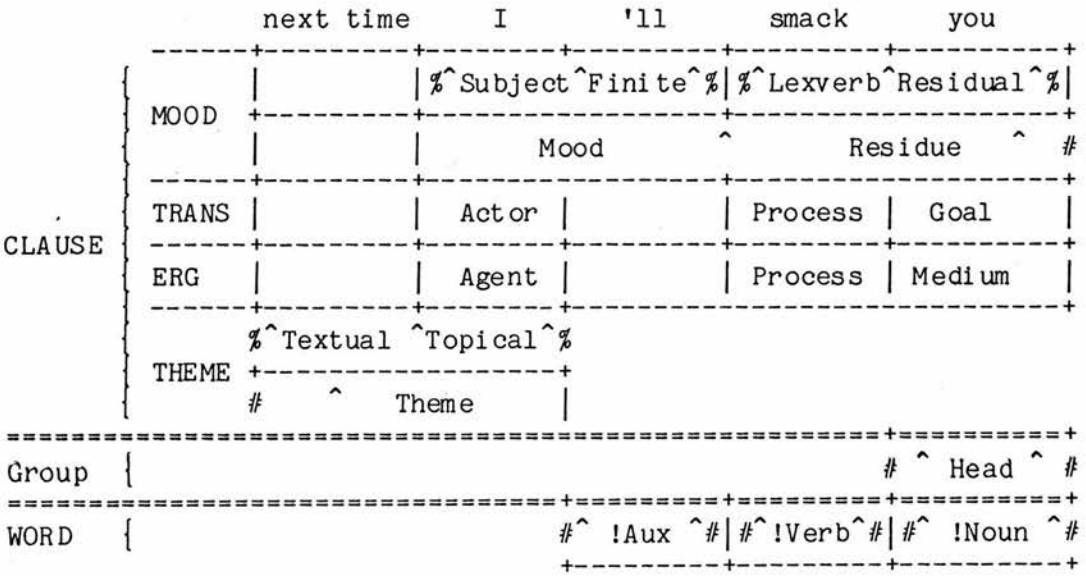


Figure 4.10

The structure of the second clause is:



Thus the text "you mustn't do that[,] next time I'll smack you"

is generated.

4.4. Advantages

This chapter began by pointing out the fundamental relationship between AI problem solving and systemic grammar. It was shown that the linguistic interpretations of the concepts found in systemic linguistics can be conflated with problem-solving interpretations. The system networks found in the linguistic literature can be interpreted as sets of forward- and backward-chaining production rules forming a linguistic knowledge base; the semantic stratum as described in the systemic theory can be interpreted as compiled knowledge--rules capable of high-level inferential macromoves; the process of generating text from systemic grammars (given the interpretation just mentioned) can be interpreted as a straightforward application of general purpose AI problem-solving techniques and domain specific knowledge to solve problems in that domain; the notion of behavioural potential can be interpreted as being described by the sum total of a problem solver's knowledge--linguistic + non-linguistic.

The fact that these two (until recently) independent fields can be conflated in this way is of theoretical and historical interest. There is, however, also an important practical significance, in that the conflation allows text generation from linguistically formalized grammars to exploit state-of-the-art AI problem-solving techniques. To date, text-generation systems have had to either pay a computational price for the use of an established linguistic formalism, or else pay a linguistic price for access to the best computational techniques (see Chapter 8). Since representations in systemic grammar and AI problem solving can be conflated, no sacrifices, linguistic or computational, need be made.

What makes text generation a hard problem (or an AI problem) is that there is a huge space of alternatives that must be discharged quickly. The discipline of AI problem solving has developed representations (e.g. production rules) for knowledge of the

alternatives, and selective, efficient techniques for searching through the space of alternatives given suitably represented knowledge. The conflation of systemic grammar with problem-solving knowledge means that these sophisticated computational techniques can be exploited while the grammar (including the semantics) retains its linguistic status.

It is not surprising that systemic grammars can be processed computationally to generate text--what is significant is that unadulterated systemic grammars, as they appear in the linguistic literature, can be processed directly by the state-of-the-art knowledge-based problem-solving methods.

It is also not surprising that AI problem-solving techniques can be used to process a linguistic formalism--what is significant in this case is that given enough knowledge at the semantic stratum--note that this is within the linguistic framework--the search is optimal in the sense that it is deterministic; only steps that lead directly to a solution are taken. Thus best-case performance is achieved for the most efficient computational techniques available.

The fundamental relationship between AI problem solving and systemic grammar will ensure that SLANG retains these advantages. As better computational techniques become available, they will still be techniques for constructing a solution "selectively and efficiently from a space of alternatives." And as better systemic grammars are developed, they will still represent language as "sets of options or alternatives."

5. Theoretical Issues

The last chapter has described SLANG, a new approach to text generation. Before going into the details of a formal model or the implementation, it may be useful to pause at this higher level of abstraction and look at the theoretical issues raised by this approach. The first set of issues is concerned with the interface between the semantics and the grammar. The interface employed in this method differs from current approaches in that parts of the semantics and pragmatics, usually dealt with by a text planner, are "compiled" to form a "semantic stratum." Another set of issues concerns taking a functional approach to computational linguistics. An analogy will be drawn between linguistic description and biological description, and some insights into the functional approach will be sought through a comparison with physiology. Finally, some important differences between the approach taken by SLANG and the approach taken by the generative paradigm will be explored. This chapter is not meant to be a presentation of concrete conclusions, but is rather an attempt to share with the reader some of the intuitions and insights which resulted in or resulted from this work.

5.1. The interfaces

The issue of compiled knowledge in SLANG introduces some interesting complications regarding the interfaces between the text planner and the text generator, and within the text generator between the semantics and the grammar. The specific issues discussed in this section will be the floating boundary between the text planner and the text generator, and the separation of semantic and grammatical knowledge.

5.1.1. The planner/generator boundary

The previous chapter initially introduced the hypothetical idea of actually doing goal-directed backward-chaining with knowledge at the grammatical stratum. It was later shown that the efficiency of this process can be greatly improved by using compiled knowledge.

Note that the boundary between the text planner and the text generator depends on how much of the possible reasoning at the grammatical stratum has been compiled, and the degree to which it has been compiled. In the case of registers where there is little or no compiled knowledge available the text planner will have to reason down to a level of detail where it can set goals involving the syntactic functions (e.g. make the Agent the Theme), which can then by solved with knowledge from the grammar. If there exists highly compiled, large-grain-size semantic knowledge applicable to the text planner's goals, it can give the text generator relatively high-level situational and social goals.

This relationship between text planning and systemic text generation should not be surprising. Text planning has always been centred around the theory of speech acts--the idea that utterances can be treated like other physical actions. This is reflected in the term "text planning" and in the methodology of planning sequences of linguistic actions using AI planning techniques usually applied to robot actions (e.g. Sacerdoti, 1975). Functional linguistics takes the same view. As Halliday says, meaning potential is just one form of behavioural potential, or "'can mean' is one form of 'can do'" (deJoia and Stenton, 1980, #549).

Note also that it is misleading in this context to view the text planner and text generator as separate mechanisms. It should be envisaged that the same inference engine is active in each case. When it is using text planning knowledge it is the "text planner" and when it is using knowledge from any of the systemic strata it is the "text generator."

Thus the boundary between the text planner and the text generator, or perhaps more accurately, the interface between the text planning and the text generation, depends directly on the status of the semantic stratum. It depends on the degree of compilation and the particular registers involved. Thus, while the distinction between text planning and text generation (or whatever terms are used in their stead) may indeed be useful, the line between them can

not and should not be drawn too heavily.

5.1.2. Separation of semantic/pragmatic and grammatical knowledge

One theoretical goal which is almost universal among computational linguists is that of keeping the semantic/pragmatic knowledge distinct from the grammatical knowledge. For the sake of modularity, neither level should contain detailed knowledge of the other. At first it may appear that the compiled nature of the semantic stratum in SLANG has violated this constraint: the preselection paths (see Section 3.3.7.5) contain what are essentially structural descriptions. These, it could be objected, belong in the grammar, not in the semantics.

The objection can be answered in two ways. First, the structural descriptions appear in the realization rules at the semantic stratum. The realization rules relate the semantics downward to the grammatical stratum, and thus should be thought of as being between the two strata. This is not just splitting hairs; there is an important point here. The semantic stratum represents the paradigmatic organization of register. The representation of register is completely independent of grammatical concerns. The realization rules embody the knowledge of how to map the various registers onto the grammatical organization.

The second part of the answer is that the semantic realization rules should contain constituency information. Immediate constituency grammars often contain constituency introduced for grammar writing convenience (e.g. recursive representation of parataxis--see Section 3.3.9). There is no doubt that the semantics should not know about this constituency. However, the minimal bracketing principle in systemic grammar (see Section 3.3.6) ensures that this form of constituency does not appear in systemic grammars.

> Literally interpreted, the wording 'minimal bracketing' would presumably mean no bracketing at all. It does not mean that, of course; what it does mean is functional bracketing--bracketing together only those sequences that have some function relative to a larger unit. (Halliday,

1985, p. 24)

Groupings can be introduced for grammar-writing convenience through the "expand" realization rule (see Section 3.3.7.2), but expanded functions never appear in preselection paths.

The preselection paths explicitly represent the hierarchy of functions formed by the minimal bracketing, and there should be no objection to this functional hierarchy appearing at the semantic stratum.

5.2. The functional approach

A set of issues not independent of the compilation issues has to do with the functional approach SLANG inherits from systemic grammar. "Function versus form" has been discussed at length in several fields of study outside AI. First, the idea of a functional approach to linguistics will be introduced, and this will be compared to the currently dominant "formal" approach.* Second, an analogy with biology will be drawn to make this comparison and to put the linguistic issues into perspective. Third, still using the analogy, the relevance of the functional approach for computational linguistics will be suggested.

5.2.1. What is a functional approach?

> The functional description of a mechanism says what it does. The implementation description says how it does it. The implementation description of the frame of which it is a part says what it is used for. If you want to understand in more detail, then the interface information, represented either explicitly or inherited from the implementation model, tells you how the functions of the parts come together to implement the total behaviour. If you want to understand in still more detail, you recurse and

* Unfortunately, the two candidates here: "formal" and "structural" are both ambiguous. "Formal," as used by Halliday (1985) and Leech (1983) seemed the lesser of the two evils. Note that this does not mean the functional approach cannot be "formalized," only that it does not deal with form as the primary characteristic of language.

> examine the functions of the subparts in the same way. Etc. ad infinitum. (Smith, 1978, p. 31)
>
> ... we are taking a functional view of language, in the sense that we are interested in what language can do, or rather in what the speaker, child or adult, can do with it; and that we try to explain the nature of language, its internal organization and patterning, in terms of the functions that it has evolved to serve. (Halliday in deJoia and Stenton, 1980, #193)
>
> What do we understand by a 'functional approach' to the study of language? ... Among other things, it would be helpful to be able to establish some general principles relating to the use of language; and this is perhaps the most usual interpretation of the concept of a functional approach. (Halliday in deJoia and Stenton, 1980, #191)

A functional approach to any domain involves classifying and relating the entities of that domain by their function. The function of an entity is "what it does," which is intimately related to "what it is used for," and when we have a hierarchy of functional entities, this becomes "how it works."

This type of functional approach has received very little attention in mainstream linguistics to date. Instead, a primarily "formal" approach has been taken--classifying and relating the entities of a domain by their structure.

5.2.2. Formal and functional approaches

The functional and formal approaches, and their relationship to each other, may best be illustrated with an example. Both the functional and formal approaches can be found in biology: physiology and anatomy respectively.

Anatomy (here referring to "regional anatomy") is the study of the structure of an organism. The organism is divided up into structural regions, which then contain substructures, and so on. This is analogous to the formal approach in linguistics where the description consists of dividing a sentence (for instance) into a noun phrase and a verb phrase, then dividing these etc.

Physiology is a functional approach because it is concerned with the functions of various components of an organism and how they combine to form larger functional wholes. Physiology is analogous to the functional approach in linguistics, represented here by systemic grammar.

> Functional description is a basic cognitive tool. Consider the problem of writing a 'grammar for animals'--a formalism that describes what their pieces are and how they fit together. In doing this, biologists look for functional systems, such as the skeletal system, the muscular system and the circulatory system. Individual structures are then described in the context of these different systems and the roles they play in them. Indeed, anatomy could be studied without any reference to this 'physiological' level of description. An animal can be seen as a complex interweaving of cords, tubes, bones, fibers etc. But to do so would make the structure seem impossibly complex and arbitrary. The key to understanding the complexity of the structure lies in recognizing its functional organization. The old adage that 'form follows function' can serve as a framework for understanding language. (Winograd, 1983, p. 279)

An analogy can be drawn between the organs of a human body (e.g. the heart) and units of a text (e.g. Subject). Each plays a functional role in the respective description, i.e. they <u>do</u> something.

It must be understood that it is important to distinguish between a functional approach, as described above, and a formal approach that uses functional labels for constituents. It is possible (and sometimes useful) to give an anatomical description (in terms of regions and spatial relationships) where the components are given functional labels such as "pump" without relating these labels to higher-level functions. It is also possible (and sometimes useful) to give formal linguistic descriptions where constituents are given functional labels such as "Agent," without relating these labels to higher-level functions.

The currently popular functional notations--in particular "functional unification grammar" (Kay, 1984, 1985)--confound this distinction because they are not explicitly associated with a

functional theory. They can thus be used with a proper functional theory, or used for formal description, or (as in the case of some recent language generation work--e.g. Appelt, 1983; McKeown, 1982, 1983) used functionally but within an ad hoc and piecemeal functional framework. What distinguishes systemic grammar from mere functional notations is the fact that it is explicitly associated with an established functional theory. All of this is analogous to biological description.

> A simple structural analysis would be based on laying out maps of organs, what they connected to, and what tissue structures appeared in which areas. A functional analysis would involve a study of physiology, viewing the body as an intertwined set of systems (such as the circulatory system and the respiratory system) and describing individual organs and internal structures in terms of the functions they serve in each of these systems. A macro-functional analysis would include an understanding of the functions these systems serve in preserving the individual and the species. The organs and structures can be described in terms of the way they contribute to one or more of the necessary macro-functions (which have been succinctly characterized as 'feeding, fleeing, fighting, and reproduction'). (Winograd, 1983, p. 288)

A structural (formal) analysis in linguistics would be based on laying out maps of constituents, what they are adjacent to, and what constituents appear in which places in the string. A functional analysis would involve viewing the text as intertwined sets of syntactic functions (such as Agent, Process, Goal; Subject, Finite; Theme, Rheme) and describing individual constituents in terms of the functions they serve in each of these. A macro-functional analysis--Halliday now uses the term "metafunctional" for adults--would include an understanding of the functions these sets of syntactic functions serve in communication. The constituents and structures can be described in terms of the way they contribute to one or more of the necessary metafunctions (which have been succinctly characterized as "ideational, interpersonal and textual").

Now, in biology the complementary relationship between the formal and functional approaches is understood.

> The distinction between anatomy and physiology as areas of study is largely a matter of emphasis; obviously the form and structure of an arm, an eye, a hand or an internal organ cannot be fully explained without reference to the functions with which these organs evolved. (Encyclopaedia Britannica, 1964, Vol. 1, p. 866)

And as Purves (1985, p.41) points out,

> One of the deepest and most productive concepts of biology is that of the intimate relationship between structure and function. To understand a structure, whether that structure is an infinitesimal protein molecule, a two meter tall termite hill, or the megalithic array of Stonehenge, we must understand its function ... In AI, one should not just study a behaviour in and for itself, but should take it in a larger context, considering its function in accomplishing goals. Natural language processing by humans relates to such goals as information transfer, and this consideration should inform research on language.

Despite all this, the functional approach has been neglected in most linguistic and (of particular interest here) computational linguistic work. As Halliday points out (1978, p. 17), despite the complementary nature of the formal and functional approaches, they "have tended to become associated with conflicting psychological theories and thus to be strongly counterposed." The next section will outline the importance of a functional approach for computational linguistics.

5.2.3. A functional computational approach

> physiology [fizi-ology] n study of the functions of and processes in living bodies. (The Penguin English Dictionary, 1979)

The quotation indicates that biologists have found it useful to study process in a functional framework (physiology) rather than in a formal framework (anatomy).

It is interesting to note that a functional approach would immediately alleviate two of the three limitations of the computational paradigm listed by Winograd (1983, pp. 28-29). First, consider the study of the social aspects of language:

> ... for example, we want to understand why a particular dialect is adopted by some members of society but not others, or how dialect differences play a role in establishing and maintaining group identity and cohesiveness. At the individual level, we may want to understand how linguistic devices serve to establish personal power relationships or to reinforce social distinctions of rank and status.

Halliday (1978), for instance, devotes an entire chapter to one extreme form of dialect variation--antilanguages* (pp. 164-182). He says, "the significance for the social semiotic, of the kind of variation in the linguistic system that we call social dialect, becomes very much clearer when we take into account the nature and functions of antilanguages" (ibid., p. 179).

> [In] the Calcutta underworld language we find not just one word for 'bomb' but twenty-one; forty-one words for 'police', and so on.... A few of these are also technical expressions for specific subcategories; but most of them are not--they are by ordinary standards synonymous, and their proliferation would be explained by students of slang as the result of a never-ending search for originality, either for the sake of liveliness and humour or, in some cases, for the sake of secrecy. (ibid., p. 165)

Note also that the study of "how linguistic devices serve to establish personal power relationships and reinforce social distinction of rank and status" is included in the study of tenor (see Section 3.5.2).

Another limitation of the computational paradigm given by Winograd (1983, p. 29) is the study of the historical aspects of language:

> ... some of the earlier paradigms for language study emphasized the historical side of language--the ways that languages evolve, divide, and merge. (Winograd, 1983, p. 29)

The antilanguages mentioned above are an example of languages

---

* An "antilanguage" is a dialect purposely developed by a group (e.g. criminals, students, comics etc.) intending to distance itself from the mainstream society for one reason or another.

dividing, and even the harshest critics of the functional approach admit a functional approach is important for the study of linguistic evolution:

> It is difficult to say what "the purpose" of language is, except, perhaps, the expression of thought, a rather empty formulation. The functions of language are various. It is unclear what might be meant by the statement that some of them are "central" or "essential".
>
> A more productive suggestion is that functional considerations determine the character of linguistic rules. Suppose it can be shown, for example, the [sic] some rule of English grammar facilitates a perceptual strategy for sentence analysis. Then we have the basis for a functional explanation for the linguistic rule. But several questions arise, quite apart from the matter of the source of the perceptual strategy. Is the linguistic rule a true universal? If so then the functional analysis is relevant only on the evolutionary level; human languages must have this rule or one like it. Suppose, on the contrary, that the linguistic rule is learned. We may still maintain the functional explanation, but it will now have to do with the evolution of English. That is, English developed in such a way as to accord with this principle. In either case, the functional explanation applies on the evolutionary level--either the evolution of the organism or of the language. (Chomsky, 1980, pp. 230-231).

It would be possible to view biological processes as causal epiphenomena of structures--processes following "a complex interweaving of cords tubes, bones, fibers etc."--but to do so would make the processes seem "impossibly complex and arbitrary." To avoid this, processes are studied in physiology where they can be organized and explained by what they accomplish. This kind of explanation--explaining something in terms of its purpose--is called teleological explanation (see von Wright, 1971), and is intimately linked with a functional approach.

A teleological explanation of the heart, for instance, would not begin by describing the subcomponents and their structural characteristics. Instead, it would be pointed out that the heart is part of the circulatory system, whose function is to provide transportation to all areas of the body, and the function or goal of the heart itself is to pump the transportation medium (the blood)

through the system. The explanation would relate downward by describing the functional subgoals of the heart (e.g. some process for creating pressure is needed).

The same reasons that process is studied in a functional framework in biology also hold for linguistics. This suggests that computational linguistics, where process is a central issue, could benefit from a functional approach. The functional approach is especially important in the social sciences because teleological explanation allows phenomena to be related upward to the social levels, something which is awkward at best in a causal framework (see Downes, 1984, Chapter 11).

5.2.4. Explanation and compiled knowledge

One of the characteristics of systems that use compiled knowledge is that some of the goal-directed reasoning is suppressed. Simply following the chain of reasoning does not provide teleological explanations with a smaller grain size than the knowledge actually used (see Brachman et al., 1983, p. 44). Nevertheless, for the most part the interest here is not in automatically providing explanations for the results of text generation, but merely to provide a framework for understanding the reasons for particular linguistic choices made during, for instance, automatic explanation in an expert system. For a helpful analogy here, the study of biology is revisited.

Good examples of compiled problem-solving strategies which are described functionally are provided by physiology. The same physiological entities that were used as examples in the earlier section on function, the heart and the circulatory system, are compiled genetically. The body, when it is developing in the womb, does not reason that it needs to circulate its blood and therefore that it needs some sort of pumping mechanism. This and other functional goals have been achieved through natural selection and the solutions have been compiled into the genetic code. As physiology demonstrates, compilation does not affect the ability to give

teleological explanations--one can still refer to the purpose of the heart or circulatory system.

Similarly, functional, teleological explanations can be provided for generated text, even though large parts of the reasoning process were compiled. A similar argument applies to the explanation of language production as a whole. Consider the discussion of linguistic explanation from Thompson (1977):

> The only explicit formulation I have been able to arrive at is one which says that to explain a phenomenon, you describe a system which in some way exhibits that phenomenon when it operates. As an example, consider the solar system. Copernicus explained the motion of the planets by describing a mathematical system which would generate that motion. Newton explained that mathematical system by describing the operation of the force of gravity. Modern theoretical physicists seek to explain the force of gravity by describing the behaviour of sub-atomic particles, ... ad infinitum.
>
> In my case then, I am trying to explain at least some aspects of the structure of English by describing a system which produces English.
>
> ... Functional explanations have been pushed back along some dimension, and I am coming to understand a little better how sense can be made of Chomsky's (1975 [see also the quotation in Section 5.2.3]) claim that functional pressures may shape language development, but do not enter directly into the mechanisms of language use, at least not at the syntactic level.

Now, if it is conceded that functional pressures are not direct, but compiled, then the situation becomes analogous to biology. If a functional approach has been taken, then, as in physiology, the explanation can be carried one step further by explaining the processes and structures by describing the goals which they serve to achieve. Notice that this does not affect the ability to give causal explanations--the system still produces English and the physical mechanism can be reduced to the level of sub-atomic particles. Thus the teleological explanation is a bonus resulting from the functional framework, compiled or otherwise.

It is important to note this position is independent of the

debate over the mechanism of the linguistic compilation itself. Traditionally, functional grammarians tend to argue that linguistic knowledge is learned (compiled cognitively), whereas generative grammarians argue that the linguistic knowledge is largely inherited (compiled genetically) (see Leech, 1983, p. 46; and Halliday, 1978, p. 17). The point made in this section remains valid whether the compiled solutions are a result of nature, of nurture, or both.

Now, in the case of SLANG specifically, the macro/metafunctional framework facilitates teleological explanations of grammatical choices, the choices that made them and the structures that realize them--even when the process is highly compiled. When a semantic feature preselects a grammatical feature, the theory indicates the broad function of that grammatical feature and therefore the high-level reason for the choice. If the feature is in the "mood" section of the clause network, for instance, the theory indicates that the contribution the feature makes is interpersonal. The theory, then, provides general guidelines for which grammatical features achieve which kinds of goals, in the same way as physiological theory provides general guidelines for which processes and structures achieve which kinds of goals. These guidelines are made specific when the relationship between register and metafunction is spelled out (see Section 3.5.4).

This section has explored the relationship between functional and formal description. It was pointed out that the complexity and arbitrariness of linguistic description can be reduced by complementing formal description with functional description. It was observed that such functional description would help fill two important gaps in the current computational paradigm: the study of the social aspects of language and the study of the historical side of language. It was also emphasized that a functional approach--even if it is complementing a formal approach--provides the basis for a teleological explanation of the processes and structures involved in computational linguistics, in exactly the same way as the functional approach to biology provides the basis for teleological explanations of biological processes and structures.

5.3. Contrasts with the generative paradigm

The approach described here has some very important differences from the generative paradigm. This section will raise two of the most important issues: Chomsky's modularity hypothesis, and the power of the grammar. It happens that these issues are related, as will be discussed below.

5.3.1. Chomsky's modularity hypothesis

One of the characteristics of the generative paradigm is that the language mechanism is viewed as being to a large degree independent of the rest of the cognitive mechanism. This view contrasts sharply with SLANG, which relies on the same problem-solving mechanism which does non-linguistic applications problem-solving to do the text generation at all levels as well. The latter approach corresponds to the computational paradigm described by Winograd (1983, p. 21):

> Most researchers in the computational paradigm give a good deal of attention to the interaction between linguistic and non-linguistic knowledge. It is assumed that the knowledge structures and processes for dealing with language are to a large degree shared with other aspects of intelligence. In developing computational models, researchers emphasize the commonalities between language and other faculties. In the generative paradigm, in contrast, it is generally assumed that there is a distinct language faculty, possessed by humans but not by other animals, which determines the structure of language.

This discussion is continued (ibid, p. 186):

> One of the central methodological principles of transformational grammar is the autonomy of syntax. It is assumed that the human mental capacities underlying syntactic competence are the result of specialized mechanisms (the language faculty) that are ... to a large extent distinct from the rest of our mental processes.
>
> ... Within artificial intelligence, on the other hand, the goal has been the identification of the general principles and mechanisms that underlie all thought processes. Instead of looking for specialized faculties (except in obvious cases like the processing in the retina), researchers formulate general theories of representation

and problem solving that can be applied to a wide range of information processing activities.

This position also matches part of Halliday's description of the "environmentalist" position--to which Halliday evidently ascribes:

> ... what the child has is the ability to process certain highly abstract types of cognitive relations which underlie (among other things) the linguistic system. (Halliday, 1978, p. 17).

This illustrates yet another dimension of the fundamental relationship between systemic theory and AI problem solving upon which this research is based.

5.3.2. The power of the grammar

One theoretical issue related both to the modularity hypothesis above and to the functional approach is the issue of the power of the grammar. Generative grammarians criticize functional grammar on the grounds that it is too powerful. But, as Winograd (1983, pp. 187) explains:

> The computational paradigm grew out of computer programming, and the formal structures are seen as analogous to programming languages and data structures. They are good to the extent that they make it possible (and easy) to model the observed phenomena of language use. Often a particular mechanism will be preferred because it is more powerful than its predecessors, since this allows it to cover more of the phenomena.
>
> In transformational grammar, on the other hand, the intuitions grew out of mathematics and the physical sciences. A theory is not something to be 'programmed,' but a terse set of axioms that can be used to predict the data. One theory is better than another because it is more restrictive, or as often stated less powerful. Much of the meta-theoretical discussion in transformational grammar deals with questions of how to limit the power of the formalism so that it can serve as a more precise theory of the capacities it models. From this point of view, most AI models are hardly theory at all.

The two issues, the modularity hypothesis and the power of the grammar, are closely related in the following way. If the

modularity hypothesis is ignored, i.e. if the same techniques are used to do general problem-solving tasks as well as language, then clearly these techniques must be very powerful. The goal of giving as simple and terse a description as possible applies to the knowledge base--in this case the grammar--not to the mechanisms or representations themselves.

5.4. Summary

This chapter has tried to throw some light on some of the important theoretical issues arising from this particular approach to text generation. The issues fell into three main categories: those resulting from the knowledge compilation, those resulting from the functional approach, and those which were issues simply because they were different than the generative approach.

The first set of theoretical issues stems from the compiled knowledge in the semantic stratum. One result of this is that there can be no firm boundary between text planning and text generation. Another result is that constituent trees appear in the semantic stratum. These, however, appear only in the realization rules and carry semantic and not syntactic information.

Another set of theoretical issues involves the functional approach to computational linguistics. Besides facilitating the study of social and historical aspects of language, the functional approach reduces the arbitrariness and complexity of linguistic description and allows teleological explanation of linguistic structures and processes.

The final area of interest here has been the relationship of this approach to generative linguistics. The two primary issues at stake are the modularity hypothesis and the power of the grammar. These are related since if, following the computational paradigm, the modularity hypothesis is abandoned then the techniques used to process the grammar will be far too powerful to be acceptable to the generative school.

In conclusion, SLANG differs from the current linguistic orthodoxy on several important theoretical points. The differences are a result of working in the computational paradigm as described by Winograd (op. cit.). In this paradigm powerful computational techniques, in this case goal-directed problem-solving with compiled knowledge, are used to process language. This results in different techniques being applied, and different theoretical assumptions being made.

6. The Formal Model

Despite the fact that systemic grammar has a relatively long history, and has been adopted in several computer implementations, it has never been given the type of rigorous formalization that traditional grammars have received. The reason for this seems to involve the difficulty of formalizing the aspects of language that are of interest to systemic linguists--the available formal tools were developed for structural description. As Winograd (1983, p. 278) comments:

> Since systemic grammar is not centered on concern with formal rules, the general attitude is that it is better to say something less precise about an important aspect of language than to ignore it completely because it does not yield to available formal tools. It is possible to provide descriptions that are structured (i.e. they include formal representations like system networks not just descriptive text) but that are not generative in the strong sense of providing rigorous rules. Much of systemic grammar follows this course.

The formal language used by systemicists to describe natural-language, like natural-language itself, reflects the functional constraints imposed upon it:

> The consumers of systemic grammar have more often been practical language teachers and sociologists than psychologists and computer programmers. The formal mechanisms were designed to be used by human interpreters, so the rules for applying them can call for judgement and interpretation. (ibid., p. 280).

It appears that in the eyes of systemic grammarians, "rigorous rules" are inherently structural. Rules have thus been avoided since it is necessary, from a functional perspective, to view language as a "resource."

> It has been customary among linguists in recent years to represent language in terms of rules.
>
> In investigating language and the social system, it is important to transcend this limitation and to interpret language not as a set of rules but as a resource. I have used the term 'meaning potential' to characterize language in this way....

> [I]n the interpretation of language, the organizing concept that we need is not structure but system. Most recent linguistics has been structure-bound (since structure is what is described by rules). With the notion of system we can represent language as a resource, in terms of the choices that are available, the interconnection of these choices, and the conditions affecting their access. (Halliday, 1978, pp. 191-192)

One of the characteristics of SLANG is that it demonstrates that system networks can be interpreted as collections of production rules. Production rules in AI are treated as a problem-solving resource--they do not necessarily describe structure.

Thus, following from Chapter 4, another important conflation is made by SLANG: the conflation of rules and resource. Systemic grammar can be given a formalization in terms of rigorous rules while these are simultaneously interpreted as a resource. This has not been possible heretofore.

The remainder of this chapter is a formal model of SLANG, including an outline of a formalization of systemic grammar. Like the rest of this thesis, this formalization is largely exploratory. It is meant to investigate and illustrate the possibility of rigorously formalizing systemic grammar in terms of production rules, derivations and so on, rather than to provide the definitive formal treatment. This is not an attempt to present a general systemic model that is compatible with all other work in computational systemic linguistics--rather the model corresponds to the version of systemic grammar presented in (Halliday, 1978) and Chapter 3. It also reflects the limits of the current implementation (see Chapter 7). For instance, chaining is only done on features--never on realization relationships (as hypothetically illustrated in 4.2.2).

6.1. A formalization of systemic grammar

Formally, a grammar G is denoted by (F, Ψ, Vn, Vt, P). The symbols F, Ψ, Vn, Vt, and P are, respectively, the features, systems, grammatical functions, terminals, and productions.

Capital letters near the beginning of the Latin alphabet will be used for grammatical functions. Lower case letters at the beginning of the Latin alphabet are used for terminals. Lower case letters near the end of the Latin alphabet denote strings of terminals. Lower case letters near the beginning of the Greek alphabet denote features; and lower case Greek letters near the end of the alphabet denote feature heaps--except ξ which denotes a logical expression.

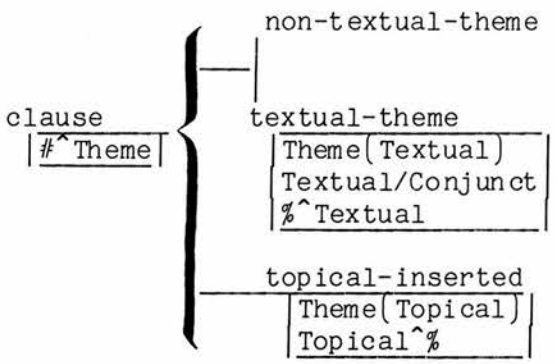
F, Ψ, Vn, Vt, and P are finite sets. F is simply the set of all features appearing in a grammar. Ψ is a set of proper subsets of F, where each of these subsets represents a mutually exclusive set of features as described in Section 3.3.2. Ψ will typically have members such as {declarative, interrogative}, {unmarked-positive, marked-positive}, {future, present, past}, {proper-noun, common-noun}, {count, mass} and so on. Since these represent mutually exclusive sets, the cardinality of the members of Ψ is always greater than one. Vn is the set of grammatical functions from all the functional analyses (see Section 3.3.5). Vn will contain members such as Agent, Theme, Deictic, Head and so on. Vt is the set of terminals or lexical items which eventually realize the constituents specified by the grammar (e.g. "the", "on the contrary", "Sir Christopher Wren", "is" etc.). Vn and Vt contain no elements in common: that is $Vn \wedge Vt = \emptyset$. The set of productions P consists of expressions of the form $\xi \rightarrow \alpha, R$ where ξ is a logical expression (an arbitrary combination of disjunction and conjunction, possibly nil) over F, α is in F, and R is a set of realization rules each having one of the following forms: A/B, A(B), A<B<...<C:β, A^B, or A=a.† (The productions represent the forward- and backward-chaining rules as described in Section 4.2.1. The logical expression ξ represents the entry conditions of feature α and R represents the realization rules [see Section 3.3.7].

† This is of course only one possible set of realization relations. The exact set of relations is not critical to the model as a whole.

The forms of the realization rules above are: conflation, expansion, preselection, adjacency and lexification.) Note that the only place elements of Vt may appear in the grammar is in realization rules of the form A=a, where aεVt (i.e. in lexify rules).

Each member α of F appears in exactly one production of the form ξ→α,R. Note that P thus implies a mapping from features to sets of realization rules.

As an example, consider the following excerpt from a clause system network (Mann/Halliday):



F = {clause, topical-inserted,
non-textual-theme, textual-theme}

Ψ = {{non-textual-theme, textual-theme}}

Vn = {Theme, Topical, Textual, Conjunct}

Vt = Ø

P = { nil→clause,{#^Theme};
clause→non-textual-theme,{};
clause→textual-theme,{Theme(Textual),
Textual/Conjunct,
%^Textual};
clause→topical-inserted,{Theme(Topical), Topical^%} }

6.1.1. Structure

Since there are several simultaneous constituent analyses (transitivity, ergativity, mood and theme), and since some of these analyses may consist of more than one layer (e.g. Mood, Residue on top; Subject, Finite, Lexverb, Residual on bottom), the structures required for this model will be more complex than those found in some of the more traditional grammars. Nevertheless, the skeleton of these structures is still a tree:

> A tree is a finite set of nodes connected by directed edges, which satisfy the following three conditions (if an edge is directed from node 1 to node 2, we say the edge leaves node 1 and enters node 2):
> 1) There is exactly one node which no edge enters. This node is called the root.
> 2) For each node in the tree there exists a sequence of directed edges from the root to the node. Thus the tree is connected.
> 3) Exactly one edge enters every node except the root. As a consequence there are no loops in the tree. (Hopcroft and Ullman, 1969, p. 18-20)

An important relationship between nodes in a tree is descendancy:

> The set of all nodes n, such that there is an edge leaving a given node m and entering n, is called the direct descendants of m. A node n is called a descendant of node m if there is a sequence of nodes n1, n2, ... ,nk such that nk = n, n1 = m, and for each i, ni+1 is a direct descendant of ni. We shall, by convention, say a node is a direct descendant of itself. (ibid.)

> A leaf node of a tree is a node that has no descendants.

Now that tree and descendant have been defined, the formal description of a systemic syntactic structure can be given. A systemic syntactic structure consists of:

1) a set N of nodes; this set is partitioned into two disjoint sets, the "unit nodes" (U) and the non-unit nodes or "expanded nodes" (X). The unit nodes are those nodes that represent unit constituents (i.e. clauses, groups, or words. See Section 3.3.6). The expanded nodes represent functional items that do not have unit status (e.g. Theme and Mood).

2) the functions OF, SUPER, MOM, LEX and ADJACENT are defined such that:

(a) OF maps from VnxU to N. (This is used to attach grammatical functions to nodes in the structure. Non-expanded functions--e.g. Subject, Topical--will be mapped onto members of U; expanded functions--e.g. Mood, Theme--will be mapped onto members of X.)

(b) SUPER maps from VnxU to members of X. (This is the formal representation of the results of the "expand" realization rule A(B)).

(c) MOM maps from N to members of U. (This mapping indicates, for each unit or non-unit functional item, the immediately enclosing unit. For instance the immediately enclosing unit of a particular item labelled "Theme" will be a particular clause.) An important subset of the MOM function defines a tree of unit nodes (U, MOM) which represents the rank consitituency of a linguistic item (see Section 3.3.6).

(d) LEX maps from terminal nodes in the tree (U, MOM) to Vt.

(e) ADJACENT is a function mapping from VnxU to Vn. The important effect of the function ADJACENT is that it implies an ordering on the leaves of the tree (U, MOM): For any two leaf nodes n,mϵN, n is ordered before m if there exist A,BϵVn and OF(A,p)=n and OF(B,q)=m and either:

a) ADJACENT(A,p) = B where p = q,

b) ADJACENT(C,r) = D, where n and m are descendants of OF(C,r) and OF(D,r) respectively, or

c) ADJACENT(A,p) = E, and OF(E,p) is ordered before m.

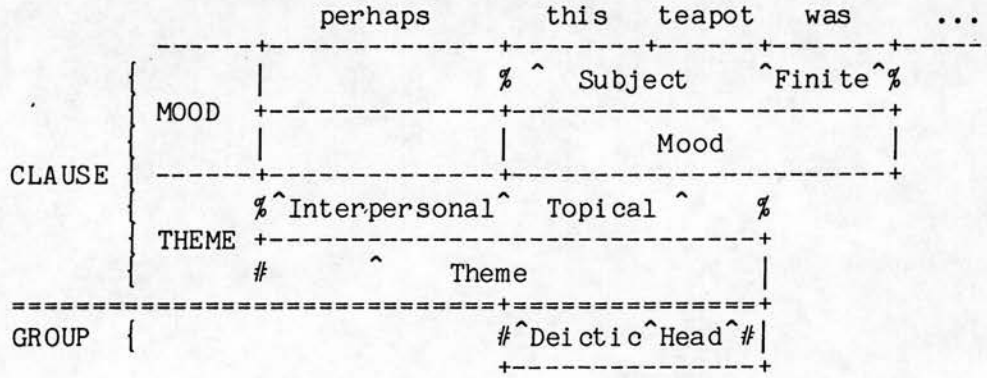
Otherwise, m is ordered before n. Informally, this is achieved by including pairs of grammatical functions from the same level of the same functional analysis in ADJACENT. That is, (Subject, Finite) can be an element of ADJACENT but not (Subject, Residue) or (Topical, Finite).

ADJACENT may operate on the "quasi-functions" # and % which indicate boundaries for members of U and X respectively. These quasi-functions can be regarded as being associated with imaginary nodes that are defined to be boundary descendants. The occurrences of the quasi-functions in the grammar are assumed to be distinct,

and for formal purposes they will be uniquely subscripted in the model.

The names of these five functions have been coined here, but some similar set of mathematical relations will be necessary for any systemic model. These functions capture formally all the relationships illustrated by the structure diagrams used in earlier chapters.

For instance, consider the following structure diagram:



This corresponds to the following partial systemic syntactic structure:

N = { n1, n2,..., n7 }

U = { n1, n2, n4, n6, n7 }

X = { n3, n5 }

OF = { ((Subject, root), n1),
((Finite, root), n2),
((Mood, root), n3),
((Interpersonal, root), n4),
((Topical, root), n1),
((Theme, root), n5),
((Deictic, n1), n6),
((Head, n1), n7) }

SUPER={ ((Subject, root), n3),
((Finite, root), n3),
((Interpersonal, root), n5),
((Topical, root), n5) }

MOM = { (n1, root), (n2, root), (n3, root),
(n4, root), (n5, root),
(n6, n1), (n7, n1) }

The LEX function involves only leaf nodes, none of which are represented in the diagram. However it would have the form: LEX = { (n12, perhaps), (n23, this), (n17, teapot), (n11, was), ...}

ADJACENT = { ((%1, root), Subject),
((Subject, root), Finite),
((Finite, root), %2),
((%3, root), Interpersonal),
((Interpersonal, root), Topical),
((Topical, root), %4),
((#5, root), Theme),
((#6, n1), Deictic),
((Deictic, n1), Head),
((Head, n1) #7) }

(The occurrences of % and # in ADJACENT are uniquely identified with subscripts.)

The result of the systemic syntactic structure is the string produced by reading the LEX values of the leaf nodes of the tree (U, MOM) from left to right.

In this case the result is "perhaps this teapot was...."

6.1.2. The language generated by a grammar

The grammar, G = (F, Ψ, Vn, Vt, P), has been presented but the language it generates has not yet been defined. The generation is defined in terms of a feature heap ω. Given the grammar G = (F, Ψ, Vn, Vt, P) and a set of unit nodes U, a feature heap from G and U is a finite subset of FxU. A feature heap thus consists of pairs: a feature and a unit node in the structure. (This is necessary to differentiate between the different nominal-group Heads in a clause, or the various Subjects in a multi-clause text, and so on).

An S-feature is a feature α such that either $\alpha \varepsilon \dagger U(\Psi)$ or there is a production $\xi \rightarrow \beta, R$ in P where β is an S-feature and α is a term in ξ. (Informally, an S-feature is either a term in a system, or directly or indirectly an entry condition of a system. S-features are those features represented in SLANG as backward-chaining rules).

† U(Ψ) denotes the "bigunion" of Ψ. I.e. if Ψ is a set of sets: {Ξ, Π, ∑, ..., Ω} then U(Ψ) = {Ξ U Π U ∑ U ... U Ω}.

Any member of F that is not an S-feature is defined to be a G-feature. (G-features represent gates--those features represented in SLANG as forward-chaining rules.)

A logical expression ξ is either a feature or a conjunction of logical expressions or a disjunction of logical expressions or nil. ξ is satisfied by a feature heap ω at node n if

a) ξ is a feature and $(\xi,n) \in \omega$.

b) ξ is a conjunction and all conjuncts are satisfied by ω at node n.

c) ξ is a disjunction and at least one disjunct is satisfied by ω at node n.

ω is a consistent feature heap of grammar G and set of unit nodes U if $\omega \subset F \times U$, and there is no $\sum$, α, β and n such that: $\sum \in \Psi$ & $\alpha \in \sum$ & $\beta \in \sum$ & $\alpha \neq \beta$ & $\{(\alpha,n), (\beta,n)\} \subset \omega$. (I.e. if ω does not contain two features from the same system at any particular node.)

A valid feature heap ω of a grammar G is a consistent feature heap such that:

1. There are no productions $\xi \rightarrow \Upsilon, R$ in P such that ξ is satisfied by ω at node n, and Υ is a G-feature, but (Υ,n) is not in ω. (There are no gates whose entry conditions are satisfied but where the gate feature has not been chosen.)
2. There are no productions $\xi \rightarrow \alpha, R$ such that (α,n) is in ω, but ξ is not satisfied by ω at node n. (There are no features in ω whose entry conditions are not satisfied).
3. For all systems $\sum \in \Psi$ such that there exists a production $\xi \rightarrow \alpha, R \in P$ where $\alpha \in \sum$ and ξ is satisfied at node n, there is exactly one feature β (where possibly $\alpha = \beta$) such that $\beta \in \sum$ & $(\beta, n) \in \omega$. (There is no system whose entry conditions are satisfied but none of whose terms appear in ω.)

Suppose S is a systemic syntactic structure, with N, U, X, OF, SUPER, MOM, LEX and ADJACENT defined as above. S is said to realize a realization rule r at node n in a feature heap ω if

a) There is some $(\alpha,n) \in \omega$ & $\xi \rightarrow \alpha, R \in P$ & $r \in R$, and

b) one of the following is the case:

r is of the form "A^B" and ADJACENT(A,n) = B. One of A or B may be a subscripted occurrence of one of the quasi-functions '#' or '%'.

r is of the form "A(B)" and SUPER(OF(B,n))=OF(A,n)

r is of the form "A/B" and OF(A,n)=OF(B,n);

r is of the form "A1<A2<...<Ai:α" and OF(A1,n)=h1; OF(A2,h1)=h2;...;OF(Ai,hi-1)=hi; and (α,hi)εω.

r is of the form "A=a" and LEX(OF(A,n))=a.

A systemic syntactic structure S is said to realize a feature heap ω if for every member of
{(r,n) | (α,n)εω & ξ→α,R ε P & rεR}, S realizes r at n in ω.

Aside from the fact that different realization relations and notation may be used, a similar treatment of realization will appear in any systemic model.

A language generated by G [denoted L(G)] is defined to be:
{z | zεVt* & ∃(S,ω).realizes(S,ω) & valid(ω) & z=result(S)}

6.1.3. Derivation

The definition of L(G) above was independent of any computational issues, and would therefore hold for systemic models in general. Traditionally, however, L(G) for some grammar is defined in terms of derivation. Although there are several ways derivation could be expressed in systemic grammar, the following corresponds to the SLANG model.

The relations => and =>* between feature heaps will now be defined. Suppose ω is a feature heap from G and U (where G = (F, Ψ, Vn, Vt, P)). If ξ→δ,R is a production in P, and n is in U,

a) ω => ω U {(δ,n)} when ω satisfies ξ at node n, and δ is a G-feature.

b) $\omega \Rightarrow \omega \cup \{(\alpha,n)\}$ when α is an atomic conjunct in ξ (i.e. ξ is a conjunction and α is a term in the conjunction) and $(\delta,n)\varepsilon\omega$, and α is an S-feature.

In each case the production $\xi\rightarrow\delta,R$ is applied to feature heap ω. a) and b) are forward- and backward-applications respectively. Thus $\Rightarrow$ relates two feature heaps exactly when the second is obtained from the first by the application of a single production.

Suppose that $\omega1$, $\omega2$, $\omega3$, ..., ωm are feature heaps such that ωi C FxU and $\omega1 \Rightarrow \omega2$, $\omega2 \Rightarrow \omega3$, ..., $\omega m-1 \Rightarrow \omega m$. Then it is said that $\omega1 \Rightarrow^* \omega m$.†† In simple terms, for two feature heaps ϕ and ω, $\omega \Rightarrow^* \phi$ if ϕ can be obtained from ω by application of some number of productions of P. By convention $\omega \Rightarrow^* \omega$ for each feature heap ω.

Note that in the case of both forward- and backward-applications, the new feature description that is added to the feature heap has the same node as those feature descriptions from which it is inferred: In the case of a) above, (δ, n) is added when, inter alia, ξ is satisfied at node n; In the case of b) (α, n) is added when, inter alia, $(\delta, n) \varepsilon \omega$. Informally, this should be expected since entry condition features should apply to the same unit as the features for which they are entry conditions (e.g. if indicative is derived from declarative these two features should both refer to a particular clause). In fact the node remains constant during a derivation (including both S-features and G-features) within any particular system network (any S-features or G-features in the clause network that are derived--however indirectly--from declarative will still refer to the same clause as declarative).

An equivalent definition of valid (defined declaratively above) could now be given in terms of derivation: A valid feature heap ω of

---

†† Say $\omega1$ derives ωm. If all the steps are forward applications, say $\omega1$ derives ωm by forward-chaining. Mutatis mutandis, $\omega1$ may also derive ωm by backward-chaining, or by a combination of forward- and backward-chaining. As the terminology indicates, this approach to derivation is specific to the SLANG model.

a grammar G is a consistent feature heap such that:

a) It is maximal with respect to =>, i.e. there is no larger consistent feature heap that can be derived.

b). There is no feature α such that $\xi \rightarrow \alpha, R \in P$ and ξ is satisfied by ω at node n, and $\{\beta \mid \{\alpha, \beta\} C \sum \& \sum \varepsilon \Psi\} \times \{n\}$ intersected with ω is $\emptyset$. (There is no system whose entry conditions are satisfied but none of whose terms appear in ω.)

L(G) could still be defined as above.

6.2. The completeness proof

A feature α is an independent disjunct if it is an S-feature and for every instance of α on the left of a production (every α in some ξ) α is a term in a disjunction.

A feature α is a tail feature if $\alpha \varepsilon U(\Psi)$ and there are no productions $\xi \rightarrow \beta, R$ in P such that α is a term in ξ and β is a S-feature. (α is a term in a system and has no systems to its right.)

A feature is a seed feature if it is an independent disjunct or a tail feature.

Lemma 1: All seed features are S-features.

Proof: If α is an independent disjunct this follows from the definition above. If α is a tail feature then $\alpha \varepsilon U(\Psi)$ and therefore is an S-feature.

For any set of nodes N, a seed is an ordered pair (α, n) where α is a seed feature and $n \varepsilon N$.

A feature α is a root feature if $nil \rightarrow \alpha, R \in P$. (I.e. if it has no entry conditions--a least delicate feature: e.g. clause, nominal-group etc.)

A feature α is less-delicate-than a feature δ if and only if either a) P contains a production $\xi \rightarrow \delta, R$ and α is a term in ξ; or b)

P contains a production $\xi \rightarrow \beta, R$ and α is a term in ξ and β is less delicate than δ.

A grammar is acyclic if the less-delicate-than relation over P is a strict partial order. (The notion of an acyclic grammar will be used to restrict the discussion to those grammars that do not contain recursive systems.)

A grammar is expressive if all root features are S-features. (Informally, this means that all system networks must have at least one system.)

Theorem: All valid feature heaps of an acyclic, expressive grammar can be derived from some initial feature heap consisting entirely of seeds.

Proof:†

Suppose there is some valid feature heap ω that cannot be derived from any initial feature heap consisting entirely of seeds. For this to be the case, there must be some element (α,n) in ω that cannot be derived from seeds. The proof will show that there can be no such element and therefore no such ω.

During the derivation of particular feature heap elements, by forward- or backward-chaining, the second part of the ordered pair (the node) remains constant. Thus instead of saying (α, n) derives (β, n) it will simply be said that α derives β. This will be called the "network assumption" since the node remains constant within a specific system network.

---

† This proof is simplified by assuming that there is an "initial feature heap" that contains not only those seeds which are given initially (as described in the discussion on derivation), but also seeds which would be preselected during the derivation. This does not affect the relevance of the proof because the origin of the seeds and the order in which they are added to the feature heap are irrelevant.

Lemma 2: Each S-feature can be derived from some seed feature.

proof: Assume that some S-feature α cannot be derived from any seed feature. If α is a seed feature this immediately results in a contradiction since by convention $\omega \Rightarrow^* \omega$. If α is not a seed feature, a contradiction is also reached as follows:

Since α is an S-feature, by definition either:

a) $\alpha \in U(\Psi)$, or

b) there is a production $\xi \rightarrow \beta, R$ in P where β is an S-feature and α is a term in ξ.

In the case of a) it follows from the definition of a seed feature and the definition of a tail feature that there is a production $\xi \rightarrow \beta, R$ such that β is an S-feature and α is a term in ξ. Thus in either case there is a production $\xi \rightarrow \beta, R$ in P such that β is an S-feature and α is a term in ξ. Further, there must be some such case where α is a term in a conjunction since it is not an independent disjunct (by the definition of seed feature). Now if β can be derived from a seed feature, then α can be derived from the same seed feature by one further backward-application.

If α cannot be derived from any seed feature then neither can β. By the same argument there must be some production $\xi 1 \rightarrow \delta, R1$ in P where δ is an S-feature and β is a term in $\xi 1$ and so on, ad infinitum. If α cannot be derived from a seed feature then this step must be applied an infinite number of times. But this results in a contradiction since the grammar is acyclic and F is finite. The original assumption always leads to contradiction, therefore each S-feature can be derived from some seed feature.

Lemma 3: All G-features Υ can be derived from seed features.

proof by minimization: Assume that there is some G-feature Υ that cannot be derived from seed features. It was stated earlier that "Each member α of F appears in exactly one production of the form $\xi \rightarrow \alpha, R$." Therefore there must be a production $\xi \rightarrow \Upsilon, R$ in P. If all terms in ξ can be derived from seed features then Υ can be derived in one additional forward application. If Υ cannot be

derived from seed features then there must be some G-feature α embedded in ξ that cannot be derived from seed features (α is not an S-feature by Lemma 2). So there is some G-feature δ in a production $\xi i \rightarrow \delta, Ri$ where all the terms in ξi can be derived from seed features but δ cannot (δ is not the root feature since the grammar is assumed to be expressive). This is a contradiction because δ can be derived in one additional forward application. Therefore, there can be no such Υ, and it follows that all G-features can be derived from seed features.

Therefore, by Lemmas 2 and 3, all features can be derived from seed features. Thus all valid feature heaps can be derived from seeds Q.E.D.

6.3. Problem reduction

The grammar G, and L(G) have been defined formally using techniques similar to those used in formalizations of more traditional grammars. At this point, the traditional computational algorithms could be given that blindly traverse the grammar and generate all the possible strings of the language. A more interesting goal here though, is to efficiently generate particular strings from the grammar. As one would expect from the model above, there are three important mechanisms: forward-chaining, backward-chaining, and structure building. The forward-chaining and structure building are simple and straight forward. The backward-chaining, since it is at the heart of this model, will be given a more substantial treatment. A formal problem-solving algorithm from the AI literature that can use the productions from the grammar for backward-chaining will be presented in this section.

This section will follow the discussion in Nilsson (1971, pp. 80-123) very closely, and the non-systemic terms and concepts have been taken from there.

The algorithm that will be used to describe the backward-chaining is called problem-reduction. The idea is to "reason backward" from the problem at hand by reducing it to subproblems, these

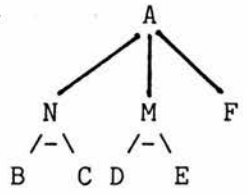
to sub-subproblems and so on, until the original problem has been reduced to a set of primitive problems that can be solved immediately.

The problem-reduction algorithm involves transforming problem descriptions into subproblem descriptions. For the text generation method here the problem descriptions will be feature descriptions. A problem description is transformed into a set of reduced or successor problem descriptions by a problem-reduction operator (ibid., p. 85). The problem-reduction operators here will be the productions in P.

The object of the problem-reduction algorithm is to eventually reduce the original problem to a set of primitive problems whose solutions are obvious. Primitive problems take one of two forms: either they are trivial problems that cannot be reduced further, or they may be other more complex problems having known solutions.

6.3.1. AND/OR graphs

Problem-reduction can be illustrated with a graph-like structure. Following Nilsson, suppose problem A can be solved either by solving problems B and C or by solving problems D and E, or by solving problem F. It is conventional to draw the graph such that each conjunction of subproblems is under its own graph node.



An AND/OR graph (ibid.)

The nodes labelled N and M are introduced as exclusive parents for the sets of subproblems {B,C} and {D,E} respectively. If M and N are thought of acting as problem descriptions, the diagram shows problem A as having been reduced to the alternative subproblems N, M or F. The graph nodes N, M and F are thus called OR nodes. Problem

N is reduced to the set of subproblems B and C and these subproblems are represented by AND nodes. AND nodes are represented by bars between their incoming arcs.

The successors of an AND/OR graph node are either all OR nodes or all AND nodes (in the case of a single successor there need be no distinction).

Terms like parent nodes, successor nodes and arc connecting two nodes will be used, and given the obvious meanings when discussing AND/OR graphs.

Each AND/OR graph has a single graph node called the start node that represents the original problem description. Graph nodes corresponding to primitive problem descriptions are called terminal nodes.

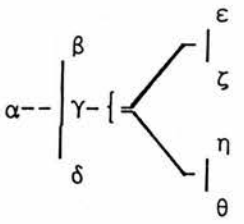
The final objective of the search of an AND/OR graph is to show that the start node is solved. A solved node in an AND/OR graph can be defined as follows:

> The terminal nodes are solved nodes (since they are associated with primitive problems)
>
> If a nonterminal node has OR successors, then it is a solved node if and only if at least one of its successors is solved
>
> If a nonterminal node has AND successors, then it is a solved node if and only if all of its successors are solved (ibid., p. 89)

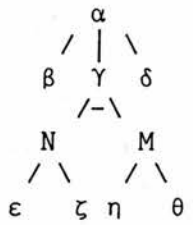
The subgraph of solved nodes that demonstrates (according to the definition above) that the start node is solved is defined to be a solution graph.

6.3.2. System networks and AND/OR graphs

It is tempting to interpret a system network as an AND/OR graph where the start node is the least delicate feature in the network. For instance, the following network where α is the root feature



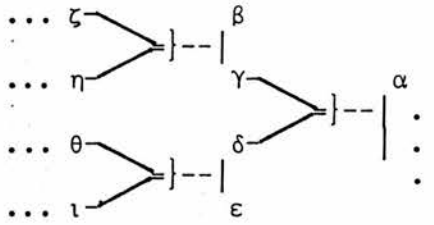
could be drawn as



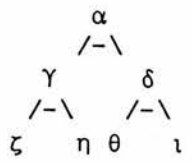
where α is the problem description being reduced. α is reduced to solving one of β, γ or δ.

This is essentially the approach taken by Nigel (Mann et al., 1983; see Section 8.4 for a discussion).

In SLANG, problem-reduction is done from seeds, so the start node in the graphs is a seed. Instead of having one large AND/OR graph stretching from left to right across the system network as a whole, SLANG has several smaller AND/OR graphs stretching from right to left where each graph is associated with an initial seed. When solving for α in a grammar where P={... (γ&δ)→α,Ri; (ζ&η)→γ,Rj; (θ&ι)→δ,Rk; ...} and where α is a seed feature

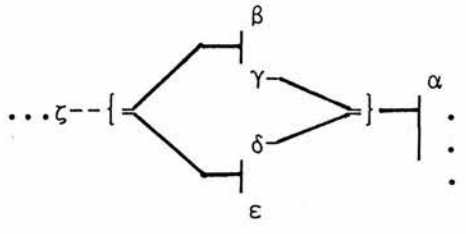


the AND/OR graph is



Note that in this case the start node is the most delicate feature instead of the least delicate. The start node α is reduced to solving both γ and δ. γ is reduced to solving both ζ and η; δ is reduced to solving both θ and ι.

The term "graph" must be used instead of "tree" because the reduction of subproblems may overlap, e.g.:

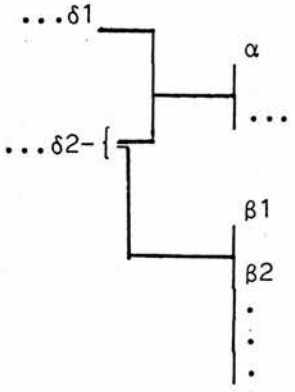


both γ and δ reduce to ζ.

In general, AND/OR graphs can be searched with techniques such as depth-first search, breadth-first search and heuristic search. When AND/OR graphs are used to represent search from seed features in systemic grammars however, they have some special characteristics.

Primitive problems in the context of SLANG are the root features since they cannot be reduced further, and any features having disjunctive entry conditions. The reason the latter are assumed to be primitive problems is that they always appear in another solution and therefore are assumed to be solved. Informally, the difficulty here is that the fact that a feature δ is a disjunctive entry condition to a problem feature α does not determine whether or not δ should appear in the solution (as opposed to the problem-reduction) of α. Given only that δ is a disjunctive entry condition to α, α can still be solved with or without δ in the solution. Thus δ must

be included or excluded from the solution on some other basis. There are two cases: a term in a disjunctive entry condition can be an independent disjunct (e.g. δ1 below), or it can be dependent on other systems (e.g. δ2 below).



In the case of independent disjuncts, there is no basis for the decision in the system network, so the only basis can be whether or not the feature is preselected--this is why independent disjuncts are seeds. All seeds are treated as initial problem descriptions and thus must have their own problem-reduction graph. Thus independent disjuncts are always part of another solution--their own solution. In the case of disjuncts that are dependent on other systems, the basis for the decision--whether or not the disjunct should appear in the solution--lies in the system network. If a disjunct δ is not an independent disjunct, then δ must be an S-feature which appears on the left of some production $\xi 1 \rightarrow \beta 1, R1$ such that δ is not a term in a disjunct in ξ. If β1 is a term in a system (as illustrated above) this implies that δ also appears as a non-disjunct on the left of productions $\xi i \rightarrow \beta i, Ri$ for all features βi in the same system as β1 (i.e. $\{Bi \mid Bi \varepsilon \sum \& B1 \varepsilon \sum \& \sum \varepsilon \Psi\}$). In this case, δ will appear in the solution graph of α if and only if some βi appears in another solution graph--in which case δ is again part of another solution.

The AND/OR graphs representing problem-reduction from seed features contain no OR nodes. This is because a problem description can never be reduced to a set of OR nodes because any feature with

disjunctive entry conditions is treated as a primitive problem and is therefore represented by a terminal node.

Lemma 4: There is exactly one reduction operator in P that can be applied to any specific problem description.

Proof: It was specified above that: Each member α of F appears in exactly one production in P of the form $\xi \rightarrow \alpha, R$. Since the productions are the reduction operators, and α represents the problem description, there is exactly one reduction operator in P that can be applied to any specific problem descriptor.

In discussions of problem solving, AND/OR graphs are usually explicitly drawn, and the search for the solution consists of trying to find a solution graph. In practice, however,

> ... we seldom have explicit graphs to search, but instead the graph is defined implicitly by an initial problem description and reduction operators. It is convenient to introduce the notion of a successor operator Γ that when applied to a problem description produces all of the sets of successor problem descriptions. (The successor operator Γ is applied by applying all of the applicable reduction operators.) (ibid., p. 90)

In the case at hand Γ will produce only one set of successor problem descriptions because by Lemma 4 there is exactly one applicable reduction operator.

Γ in the case of SLANG can be defined simply as backward applications of =>.

Lemma 5: The solution graph for an AND/OR graph where the start node is a seed feature represents exactly those features that can be derived from that seed feature by backward-chaining.

Proof: (making the network assumption) Since backward applications of => are equivalent to Γ (see above), and since the features that can be derived from a seed feature by backward-chaining are those that can be derived by some number of backward applications of =>, it follows that exactly the same features will appear in the solution graph since each of these is the result of some number of

applications of Γ.

Since there is no disjunction there is exactly one solution graph for any problem.

6.4. Algorithms

At this point computational algorithms will be given for text generation within the SLANG approach. In general terms, the derivation (forward- and backward-chaining) is done by the PROBLEM-SOLVER which is domain independent--it could be used for solving non-linguistic problems. The backward-chaining is expressed in terms of problem-reduction (see previous section). The third task that must be performed is realization. In the next chapter, which describes the implementation of SLANG-I, the realization is represented as production rules and executed by the problem solver. This is merely for convenience--realization does not require, or take advantage of, the power of the problem solver. The derivation, on the contrary, does exploit the ability of the problem solver to selectively and efficiently search through a space of alternatives. For these reasons the realization is explicitly separated from the derivation and depicted as a simple procedure which could be implemented in almost any computational notation.

```
algorithm: GENERATE (G, semantic-seeds)
input: G subfields F, Ψ, Vn, Vt, P ; a grammar
input: semantic-seeds ; a set of semantic feature
        ; descriptions where the node in each case
        ; is some identifier that by convention
        ; represents the root node of the structure tree.
variable: ω ; the feature heap
variable: OF, SUPER, MOM, LEX, ADJACENT ; functions
variable: g-set ; a local variable
begin
  g-set ← goals(semantic-seeds) ; mark seeds as goals
  PROBLEM-SOLVER(P, ω, g-set)
    ; ω here is a pointer so the feature heap can be
    ; modified by the problem solver.
end.
```

```
algorithm: PROBLEM-SOLVER (K, M, goals)
input: K ; a knowledge-base in the form of a set
         ; of production rules.
input: M ; the working memory represented as a
         ; set of facts or patterns.
input: goals ; a set of goals

concurrent procedures: FORWARD-CHAIN,
                       BACKWARD-CHAIN
  algorithm: FORWARD-CHAIN
  variable: Fdone, initially Ø
  begin
    if there is some k ε K
       and there is some set of facts f ε M
       and match(LHS(k), f)
       and not (f, k) ε Fdone
    then EXECUTE(f, k, M)
       Fdone ← Fdone U {(f, k)}
  repeat
  end.

  algorithm: BACKWARD-CHAIN
  variable: Bdone, initially Ø
  begin
    if there is a goal g in goals
       and not g ε Bdone
    then SOLVE(g)
  repeat
  end.
```

```
;;; the problem-reduction algorithm

algorithm: SOLVE (p)
input: p ; a problem
variable: successors ; a set of problems
begin
  if p ε Bdone then return
  Bdone ← Bdone U {p}
  successors ← Γ(p)
  EXECUTE(p, Γ'(p), M)
    ; Γ'(p) returns the production used for
    ; the expansion of p
  if primitive(p) then return
  if successors is a set of sets
    ; i.e. a disjunction
  then DISJUNCTION(successors)
    ; this will never happen in SLANG because
    ; the primitive check checks for disjunction
    ; and returns before this is executed
  else for each member s of successors
         goals ← goals U {s}
return
end.
```

Lemma 6: SOLVE(g) adds exactly those feature descriptions to the feature heap that can be derived from g (a feature description) by backward-chaining.

Proof: SOLVE simply traverses the solution graph for g which by Lemma 5 represents all the feature descriptions that can be derived from g by backward-chaining.

The procedure EXECUTE called by both FORWARD-CHAIN and SOLVE is domain specific--it recognizes specific patterns and acts accordingly.

```
algorithm: EXECUTE (fd, p, ω)
input: fd ; a feature description
input: p  ; a production
input: ω  ; the feature heap
          ;(a pointer to working memory)
begin
  ω ← ω U fd
  for each realization rule r in the RHS of p
    REALIZE(r, node(fd), ω)
end.
```

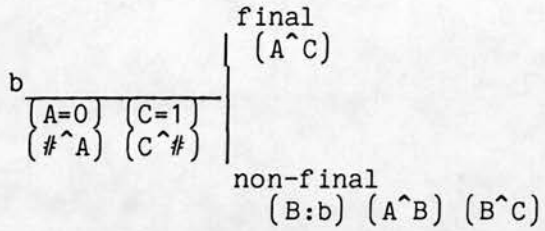
The algorithm REALIZE builds a structure according to the realization rules in the feature heap. Note that preselection modifies the feature heap--if it adds seeds, this will result in additional problem-reduction by SOLVE and so on.

```
algorithm: REALIZE (r, n, ω)
input: r ; a realization rule
input: n ; a structure node
input: ω ; the feature heap

begin

  If r is of the form "A^B"
  then
    if A = # or % then
      A ← subscript(A, make-unique-subscript())
    if B = # or % then
      B ← subscript(B, make-unique-subscript())
    ADJACENT(A,n) ← B

  If r is of the form "A(B)"
  then SUPER ← SUPER U {(OF(B,n), OF(A,n))}
```

```
If r is of the form "A/B"
then
  OF(B,n) ← OF(A,n)
  if LEX(OF(B, n)) ≠ Ø
  then LEX(OF(A, n)) ← LEX(OF(B, n))
  for all members of ω having the form (A,n),
    substitute B for A.
  for all members of MOM having the form ((A,n),m),
    substitute B for A.
  for all members of SUPER having the form ((A,n),s),
    substitute B for A.

If r is of the form "A1<A2<...<Am:α"
then
  h0 ← n
  for i ← 1 to m do :
    if OF(Ai,hi-1)=Ø
    then OF(Ai,hi-1) ← create-node;
    hi ← OF(Ai,hi-1);
    MOM(hi) ← hi-1.
  ω ← ω U {(α,hm)}

If r is of the form "A=a" then LEX(A,n) ← a.

end.
```

6.5. An example

Consider the following system network:



This system network can be represented formally as the grammar:

F = {b, final, non-final}
Ψ = {{final, non-final}}
Vn = {A, B, C}
Vt = {0, 1}
P = { nil→b,{A=0, C=1, #^A, C^#};
b→final,{A^C};
b→non-final,{B:b, A^B, B^C} }

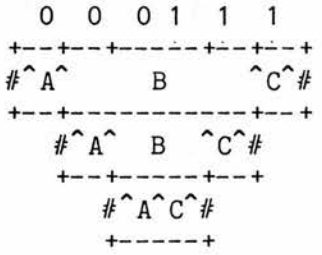
This grammar generates the well known language $\{0^n1^n | n \geq 1\}$.

The features will be referred to with paths: e.g. Subject:determined, Subject<Deictic<Deictic:possessive, Subject<Deictic<Head:singular and so on. So instead of representing members of a feature heap as ordered pairs, they are represented as a single descriptor using the path notation.

The realization rules will be omitted from the productions for simplicity.

Returning to the example, the standard left-to-right generation described in most of the systemic literature begins by initializing the feature heap to {b} (making the root feature the initial problem description). At this point some mechanism must choose between final and non-final. Suppose non-final is chosen. b→non-final => {b, non-final} generating a string 0B1 where B is a non-terminal (grammatical function). The feature B:b (a level of nesting now) is preselected as part of the last derivation step--this initializes the feature heap for the recursive generation: {b, non-final, B:b}.

Suppose again non-final is chosen. b→non-final => {b, non-final, B:b, B:non-final} with the preselection resulting in B<B:b being added to the feature heap. Suppose this time final is chosen. b→final => {b, non-final, B:b, B:non-final, B<B:b, B<B:final}. Now no more productions can be applied consistently. The result of the derivation tree is "000111".



SLANG, in contrast, backward-chains from seeds; seeds are the initial problem descriptions. Three seeds are required to generate "000111": {non-final, B:non-final, B<B:final} which together form the initial feature heap. From the seed non-final, b→non-final derives b by one backward-application. The backward-chaining stops after only one application because no more problem-reduction can be done--b is a primitive problem. From the seed B:non-final, b→non-final derives B:b (again the chain must stop here). From the seed B<B:final, b→final derives B<Bb. The final feature heap is {b, non-final, B:b, B:non-final, B<B:b, B<B:final}. This is realized by the same syntactic structure as above, and it has the same result. The significant difference is that the derivation in the SLANG model is deterministic given the original feature heap containing the necessary seeds.

7. The Implementation

This chapter will describe the first implementation of the Systemic Linguistic Approach to Natural-language Generation (SLANG-I). SLANG-I has been implemented as a production system using the production language OPS5. Since many OPS5 rules appear in this chapter, a short introduction to OPS5 has been provided in Appendix A.

This chapter is divided into five sections. The first is an overview of the system as a whole. It will provide high-level descriptions and explanations for the mechanisms used in the implementation. The second section is a detailed description of the System Network - OPS5 Rule Translator (SNORT) which outputs the grammar in the form of OPS5 production rules that can be used by SLANG-I. The text-generation system itself--SLANG-I--is described in detail in the third section. The fourth and fifth sections look at the limitations of this implementation and some possible alternatives respectively. Finally, a summary is given.

7.1. Overview

The purpose of this section is to provide a high-level overview of SNORT and SLANG-I before getting down to details in the next two sections. To a large extent this is made necessary by the interdependence between these two components. It is impossible to motivate the output of SNORT before explaining to a certain extent how SLANG-I works, and similarly SLANG-I cannot be explained before it is understood how the grammar is represented in OPS5 production rules. This section consists of four parts: first, a presentation of the abstract architecture of SLANG-I; second, a discussion of the OPS5 productions representing the systemic grammar; third, a description of the data structures used by SLANG-I; and fourth, a brief look at the control strategy used to coordinate the text-generation process.

7.1.1. The abstract architecture

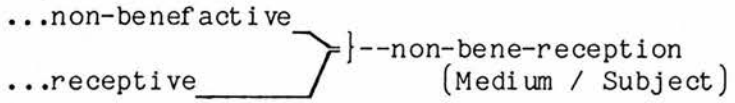
The architecture of SLANG-I is very simple. The primary components are a "problem solver" (the OPS5 inference engine slightly abstracted in the sense that it is being used to do both forward- and backward-chaining--see Section 7.1.2) and a "knowledge base" (the system networks represented as OPS5 productions). As described in Chapter 4, the problem solver simply forward- and backward-chains using the productions. As the problem-solving proceeds, the realization rules attached to those features that are inferred by the problem solver are processed to build a structure realizing the text. It was convenient to process the realization rules with OPS5 productions, so the problem solver also builds the structures. If a more sophisticated problem solver were used in SLANG (e.g. if SLANG were implemented within an expert system) it may also be convenient for the problem solver to build the structure. But note that it is not important whether or not the problem solver performs this task-structure building does not involve search so it does not require the power of the problem solver. So the realization (with the important exception of preselection, which sets goals to be solved) is not really part of the problem solving per se; it could just as easily be performed by a simple program.

Thus the abstract architecture is that there are three major components: the problem solver, the knowledge base (grammar), and a mechanism (which in this case happens to be written as productions used by the problem solver) that builds the syntactic structures and outputs the text.

7.1.2. The grammar productions

The previous chapters have already described in abstract terms the forward- and backward-chaining rules which are the SLANG interpretation of gates and system features respectively. To avoid confusion with these abstract rules, and also with realization rules, the term "production" will be used instead of "rule" when referring to OPS5 code.

The forward-chaining rules (gates) can be implemented as OPS5 productions in a very straight-forward manner. Working memory elements of the form (chosen x) can represent the fact that feature x has been chosen. The left-hand-side (LHS) of the production simply specifies the logical combination of features acting as the entry conditions of the gate. The right-hand-side (RHS) puts an element of the form (chosen ...) in working memory indicating that the feature has been chosen, and puts a "realization statement" in working memory for each realization rule of the gate ("realization statement" is used for the OPS5 version of a systemic realization rule). For instance the feature non-bene-reception



can be written in OPS5 as

```
;;;IF   the features non-benefactive and receptive
;;;     have been chosen
;;;THEN choose non-bene-reception and conflate
;;;     the Medium and the Subject
(p non-bene-reception::first-approx
  (chosen non-benefactive)
  (chosen receptive)
-->
  (make chosen non-bene-reception)
  (make conflate ^fun Medium ^with Subject))
```

Ideally, backward-chaining rules could be implemented similarly, with the entry conditions in the LHS, realization statements in the RHS, and a (make chosen ...) statement also in the RHS. Then if there was a goal to choose such a feature, the realization rules would be treated as effects, and the entry conditions would be set as subgoals. Note that this would involve looking at the RHS of productions to determine which productions satisfy the goal. Unfortunately, the OPS5 architecture--technically speaking--is strictly forward-chaining; productions are selected solely on the basis of matching the LHS with working memory.

This problem can be overcome by using working memory elements of the form (goal x) and writing productions so the goal statement

is matched in the LHS (e.g. Brownston et al., 1985, Appendix). The entry conditions must be made into subgoals in the RHS. For instance:

```
;;;IF   there is a goal to choose declarative
;;;THEN change the goal statement
;;;       to a chosen statement,
;;;     and set indicative as a subgoal,
;;;     and make the Subject adjacent
;;;       to the Finite element
(p declarative::first-approx
  (goal declarative)
-->
  (modify 1 ^1 chosen)
  (make goal indicative)
  (make adjacent ^to Subject ^is Finite))
```

Thus backward-chaining can be implemented within the OPS5 forward-chaining architecture. Notice this prevents the realization relationships being stated as goals (e.g. there can be no goal to make the Agent the Theme as in 4.2.2; the original goals are all preselected features as in 4.2.3).

Although all OPS5 productions are--in the strictly technical sense--forward-chaining, the terms "forward-chaining production" and "backward-chaining production" will be used to refer to productions that represent the abstract forward- and backward-chaining rules described in Chapter 4.

Another problem resulting from OPS5 is that several identical elements in working memory will independently match the LHS of productions. This may cause a production to fire several times, perhaps generating more redundant working memory elements. For instance, several features have declarative as an entry condition. If each one puts a separate (goal declarative) in working memory, the production above will fire several times, generating several copies of (goal indicative), which in turn will cause the production for indicative to fire several times (in addition to all the redundant firings as a resulting from it being an entry condition to several features) and so on. A similar phenomenon occurs when gates have disjunctive entry conditions. If several of the disjuncts are satisfied, the production will fire for each one--resulting in

several firings of productions whose LHS matches the identical copies of the "chosen" statement and so on.

This accumulation of redundant work for the problem solver is unacceptable. The solution to this problem adopted here is to call a LISP operator, which checks if a feature has been chosen already before it creates a new "goal" or "chosen" statement. These operators are called GOAL and ASSERT respectively and are used throughout the implementation instead of "make goal" and "make chosen" respectively.

```
;;;IF   there is a goal to choose declarative
;;;THEN choose declarative, set indicative as
;;;      a subgoal, and make the Subject adjacent
;;;      to the Finite element
(p declarative::second-approx
  (goal declarative)
-->
  (modify 1 ^1 chosen)
  (call GOAL indicative)
  (make adjacent ^to Subject ^is Finite))
```

7.1.3. The syntactic structures

The productions just described represent realization rules using OPS5 realization statements that are put into working memory. A substantial part of the SLANG-I system consists of OPS5 productions whose LHS matches these realization statements. The RHS of each of these productions modifies the structure of the text accordingly. These realization productions are described in detail later; the point that needs to be made here is that the structure being built plays an important role in the text generation.

The building block for the linguistic structures is the "hub" (from Mann et al., 1983). A hub is represented by a unique atom--generated by LISP's "gensym" or OPS5's "genatom." Recall from Chapter 6 that the structure has the form of an acyclic directed graph. The hubs are the nodes in this graph. Each hub is associated with, inter alia, a set of functions (Agent, Subject, etc.). The hubs themselves do not have individual entries in working

memory, but merely appear in the individual entries for each of the associated functions. Each function has an entry in working memory with the following template:

```
(hub ^of        [e.g. Subject, Head, !Noun etc.]
     ^is        [the hub identifier e.g. g00023]
     ^super     [another hub e.g. g00005]
     ^output?   [either yes, no, or nil]
     ^lex       [e.g. ran, John, in etc.]
     ^mom       [another hub e.g. g00007]
)
```

Note the correspondence between some of these fields and the formal description of the structure in the previous chapter.

If two functions are conflated, for instance the Subject and the Agent, the descriptions will be changed so that they have the same ^is field (i.e. they have the same hub): e.g. (hub ^of Subject ^is g00015...), (hub ^of Agent ^is g00015...). The ^super field describes hierarchies of functions as defined by the expansion realization rule. For instance the Subject and the Finite are both subfunctions of the Mood, hence the hub of the Mood will appear in the ^super field of both the Subject and Finite:

```
e.g.
(hub ^of Finite ^is g00056 ^super g00023...)
(hub ^of Subject ^is g00043 ^super g00023...)
(hub ^of Mood ^is g00023...)
```

Similarly, the ^mom field contains the hub of the next unit above in the structure hierarchy. For instance if the Subject in a clause is realized by a nominal-group, all the functions in that nominal-group will have the hub of the Subject in their ^mom field. If the Head of that nominal group is realized at the word rank by a noun, the function in the noun network (there is only one--!Noun) will have the Head of the nominal group in its ^mom field.

All this leads up to an important point: there may be several

nominal groups in, for instance, a clause, and it is not good enough to put (chosen determined) in working memory without indicating to which of the nominal groups it refers. If both the Subject and the Medium of a particular clause are realized by nominal groups, and (hub ^of Subject ^is g00016 ^mom g00005) and (hub ^of Medium ^is g00018 ^mom g00005) and the Subject is determined, then (chosen determined g00016) will unambiguously state this fact. A similar addition must be made to "goal" statements and realization statements. The final form of a forward-chaining production, for instance non-bene-reception, is:

```
;;;IF   the features non-benefactive and receptive
;;;     have been chosen
;;;THEN choose non-bene-reception and conflate
;;;     the Medium and the Subject
(p non-bene-reception
  (chosen non-benefactive <mom>)
  (chosen receptive <mom>)
-->
  (call ASSERT chosen non-bene-reception <mom>)
  (make conflate ^fun Medium ^with Subject ^mom <mom>))
```

7.1.4. The control strategy

Before the final form for backward-chaining productions can be given, one last issue must be resolved: the control strategy. It is desirable to have the text generated in a left-to-right fashion so it can be output as it is generated. This is accomplished by marking the hubs in the structure as "sub-judice" (under consideration), in a left-to-right depth-first manner, and not firing backward-chaining productions unless they are relevant to hubs in a "sub-judice" statement in working memory.

```
;;;IF   there is a goal to choose declarative
;;;      at a node under consideration
;;;THEN choose declarative, set indicative as
;;;      a subgoal, and make the Subject adjacent
;;;      to the Finite element
(p declarative
  (sub-judice <mom>)
  (goal declarative <mom>)
-->
  (modify 2 ^1 chosen)
  (call GOAL indicative <mom>)
  (make adjacent ^to Subject ^is Finite ^mom <mom>))
```

Since forward-chaining productions are dependent on features chosen by backward-chaining productions, only the latter need an explicit check.

7.1.5. Overview conclusion

This overview has provided a high-level gloss of the workings of SLANG-I and motivation for the form of the productions representing the grammar. The next section will describe in detail how these productions are generated automatically from a system network notation. The section following that will describe the details of SLANG-I including the productions for realization, and productions for implementing the control strategy and output.

7.2. SNORT (System Network - OPS5 Rule Translator)

The implementation of SLANG-I depends on the systemic grammar being in the form of OPS5 productions. To this end a set of LISP operators has been written that translates from a system-network-like notation to OPS5 productions (the System Network - OPS5 Rule Translator: SNORT). This section will briefly describe this translation and the program that performs it.

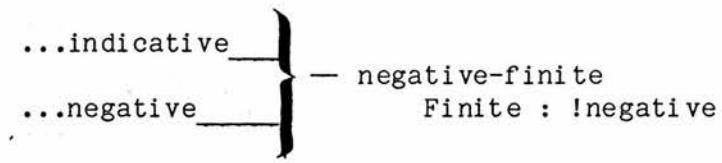
7.2.1. The system network notation

Since it is impractical to enter grammars in the graphical notation in which they appear in the linguistic literature, some notation must be used that can easily be input using ordinary

keyboards and characters. The solution to this problem is a LISP based notation which represents the grammar feature-by-feature, but which also attempts to capture some of the graphics through the use of symbols such as -{=,]-, =}- and so on. The idea was not to create a notation that the grammar-writer can use to develop the system networks, but rather to create a notation that the grammar-writer could easily use to type in his system networks once they have been developed. It is intended that the grammar-writer modifies and expands the grammar while referring to the original graphical notation, and then enters these changes using the LISP notation.

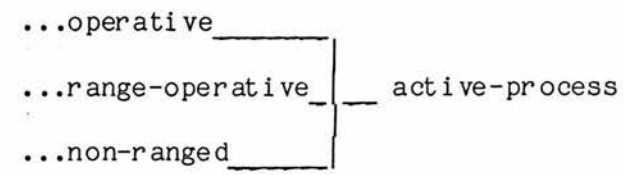
First, consider the gates or forward-chaining rules. These are represented as lists (EC f RR1 RR2 ...) where the first element of the list is a description of the entry conditions, the second element is the name of the feature, and any further elements are realization rules. The description of an entry condition is a list where the the final element is a graphical symbol and the other elements are either features or nested descriptions of entry conditions. The graphical symbols =}- and -<> represent conjunction, the symbol]- represents disjunction, and the symbols -- and -{= are used when there is only one feature acting as the entry condition.

Here are some examples:



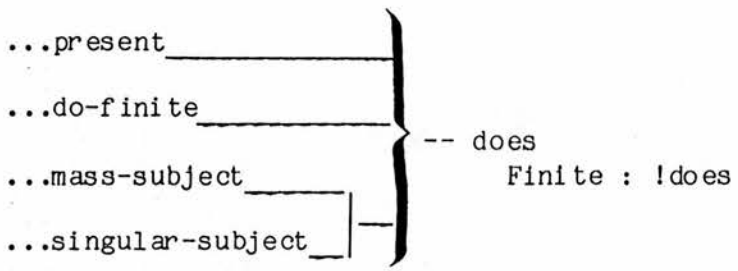
is written

```
((indicative negative =}-) negative-finite
       (Finite : !negative)).
```



is written

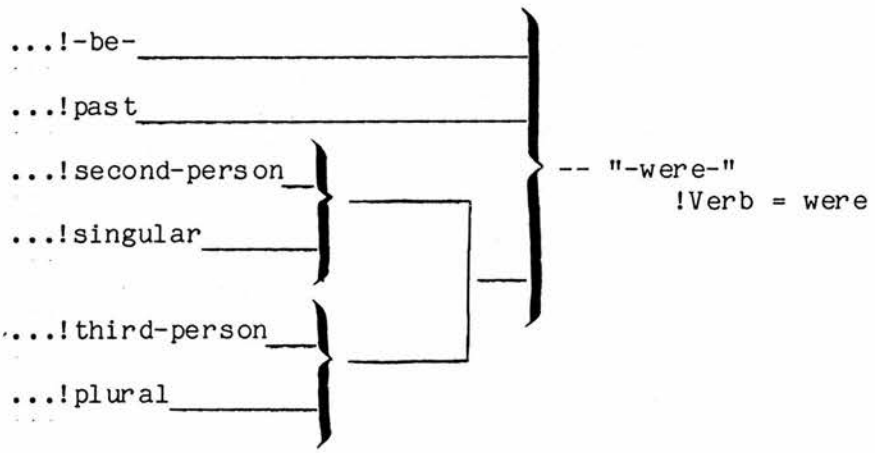
```
((operative range-operative non-ranged ]-) active-process).
```



is written

```
((present
  do-finite
  (mass-subject singular-subject ]-)
 =}-) does
     (Finite : !does))
```

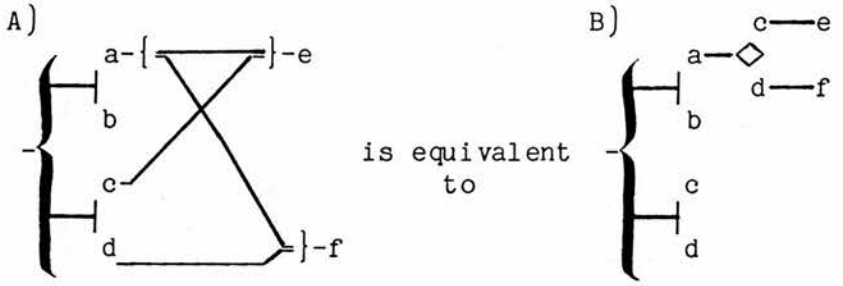
Note that the nesting can be arbitrarily complex with ordinary LISP parentheses indicating the nesting. For instance



is written

```
((!-be-
  !past
  ((!second-person !singular =}-)
   (!third-person !plural =}-)
  ]-)
 =}-) "-were-"
     (!Verb = were))
```

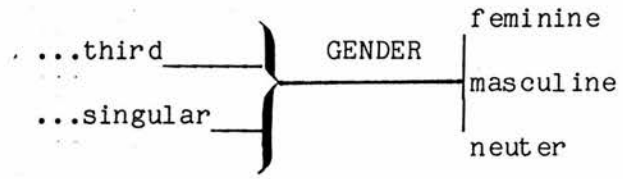
The "diamond" notation which is often used to avoid tangled system networks has also been implemented.



Part A) of the above diagram uses the bracket notation which requires crossed lines. Features e and f are represented in this case by ((a c =}-) e) and ((a d =}-) f) respectively. Part B) of the diagram uses the diamond notation to avoid the crossed lines. In this case the same features are represented by ((a c -<>) e) and ((a d -<>) f). Note that in the LISP notation =}- and -<> are synonymous.

Features that are terms in systems are represented in much the same way as features that are gates. The graphical symbol -[is used to represent a system.

As in the case of gates, complex entry conditions are simply nested. The following system from (Winograd, 1983, p. 293)



can be written as three statements--one for each feature:

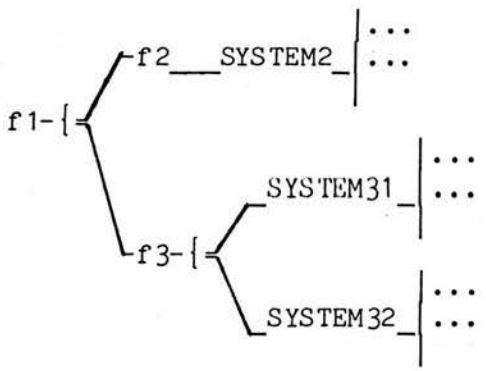
```
(((third singular =}-) -[ ) feminine)
(((third singular =}-) -[ ) masculine)
(((third singular =}-) -[ ) neuter)
```

Labelling of systems (e.g. GENDER above) is a common practice, and often provides useful documentation. Thus the notation allows

the system labels, in capitals, to appear in place of the entry condition to the system. The above system can also be written:

```
((third singular =}-) GENDER)
((GENDER -[) masculine)
((GENDER -[) feminine)
((GENDER -[) neuter)
```

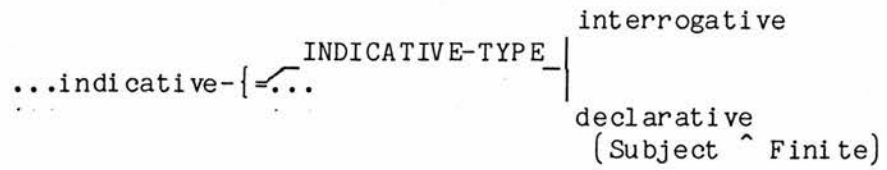
Often there is a single feature acting as a particular entry condition. The graphical symbols used to represent this are -- and -{=. The difference between these is purely visual; the latter is used if a branch appears in the system network:



This piece of system network would be written:

```
((f1 -{=) f2)
  ((f2 --) SYSTEM2)
    ((SYSTEM2 -[) ...)
    ((SYSTEM2 -[) ...)
((f1 -{=) f3)
  ((f3 -{=) SYSTEM31)
          .
          .
          .
  ((f3 -{=) SYSTEM32)
          .
          .
          .
```

As another example:



is written:

```
((indicative -{=) INDICATIVE-TYPE)
((INDICATIVE-TYPE -[) declarative (Subject ^ Finite))
((INDICATIVE-TYPE -[) interrogative)
```

7.2.2. The production rule notation

The system network notation is translated by SNORT into OPS5 (Forgey, 1981) (Brownston, et al., 1985) production rules. There are two production templates that are used, one for forward-chaining and one for backward-chaining. The forward-chaining productions have the form:

```
(p feature-name
  (chosen entry-condition1 <mom>)
  (chosen entry-condition2 <mom>)
          .
          .
          .
  (chosen entry-conditionN <mom>)
-->
  (call ASSERT chosen feature-name <mom>)
  (make realization-statement1 ^mom <mom>)
  (make realization-statement2 ^mom <mom>)
          .
          .
          .
  (make realization-statementN ^mom <mom>))
```

The entry conditions illustrated in this template are conjuncts. If there is disjunction within the conjuncts, OPS5 disjunction << d1 d2 d3 ... dn >> is used to represent this. Also, the realization statements are written in the OPS5 attribute-value notation. These

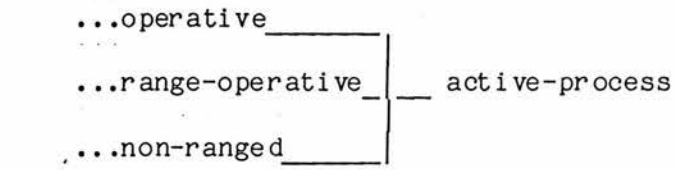
take one of the following forms (equivalent rule form in parentheses):

```
adjacent ^to Function1 ^is Function2 (F1 ^ F2)
expand ^fun Function1 ^into Function2 (F1(F2))
conflate ^fun Function1 ^with Function2 (F1 / F2)
lexify ^fun Function1 ^as lex1 (F1 = lex1)
preselect ^feature feature1 ^for Function1 ... FunctionN
 (F1<F2...<FN : f)
```

For instance the forward-chaining rule does illustrated above is written:

```
;;;IF   present, do-finite and one of mass- or
;;;     singular-subject have been chosen
;;;THEN choose does and preselect the lexical feature
;;;     !does for the Finite element
  (p does
    (chosen present <mom>)
    (chosen do-finite <mom>)
    (chosen << mass-subject singular-subject >> <mom>)
  -->
    (call ASSERT chosen does <mom>)
    (make preselect ^feature !does ^for Finite ^mom <mom>))
```

The forward-chaining rule for active-process



is written:

```
;;;IF   one of operative range-operative or non-ranged have
;;;     been chosen
;;;THEN choose active-process
  (p active-process
    (chosen << operative range-operative non-ranged >> <mom>)
  -->
    (call ASSERT chosen active-process <mom>))
```

At this point a complication is encountered. Recall the gate "-were-" above. It contains conjunction nested within the

disjunction. This was no problem for the LISP notation, but it poses a problem for the simple OPS5 templates described above. In OPS5 there is no way to express conjunction nested within disjunction in this way. The solution is not difficult however: intermediate productions can be introduced such that there is only one level of conjunction in any one production. In the case of "-were-", for instance, the problem is conjoining !second-person and !singular within a disjunct, and !third-person and !singular in another disjunct. If two intermediate productions are written (using the same template illustrated above):

```
;;;IF   !second-person and !singular have been
;;;     chosen
;;;THEN choose the dummy feature "-were-"g00005
  (p "-were-"g00005
    (chosen !second-person <mom>)
    (chosen !singular <mom>)
  -->
    (call ASSERT chosen "-were-"g00005 <mom>))
```

and

```
;;;IF   !third-person and !plural have been
;;;     chosen
;;;THEN choose the dummy feature "-were-"g00006
  (p "-were-"g00006
    (chosen !third-person <mom>)
    (chosen !plural <mom>)
  -->
    (call ASSERT chosen "-were-"g00006 <mom>))
```

then the forward-chaining production can be written as

```
;;;IF   one of the dummy features "-were-"g00005
;;;     or "-were-"g00006 have been chosen as well as
;;;     !-be- and !past
;;;THEN choose "-were-" and assign !Verb the lexical
;;;     item "were".
  (p "-were-"
    (chosen !-be- <mom>)
    (chosen !past <mom>)
    (chosen << "-were-"g00005 "-were-"g00006 >> <mom>)
  -->
    (call ASSERT chosen "-were-" <mom>)
    (make lexify ^fun !Verb ^as were ^mom <mom>))
```

Since an arbitrary number of intermediate levels of productions can be introduced (recursively, as described below), arbitrarily complex

entry conditions can be handled.

The backward-chaining productions have, of course, a different template:

```
(p feature-name
  (sub-judice <mom>)
  (goal feature-name <mom>)
-->
  (modify 2 ^1 chosen)
  (call GOAL entry-condition1 <mom>)
  (call GOAL entry-condition2 <mom>)
                .
                .
                .
  (call GOAL entry-conditionN <mom>)
  (make realization-rule1 ^mom <mom>)
  (make realization-rule2 ^mom <mom>)
                .
                .
                .
  (make realization-ruleN ^mom <mom>))
```

As one would expect, the production is activated if the particular feature is a goal, the entry conditions are set as subgoals, and the realization statements are the effects. The "modify 2 ^1 chosen" means "modify the first field of the second condition to 'chosen'" so that the "goal" statement is now a "chosen" statement. The "sub-judice" at the beginning of the production is for controlling the order of execution of the productions, and will be explained later. For instance:

```
;;;IF   there is a goal to choose declarative
;;;      at a node under consideration
;;;THEN choose declarative, set indicative as
;;;      a subgoal, and make the Subject adjacent
;;;      to the Finite element
  (p declarative
    (sub-judice <mom>)
    (goal declarative <mom>)
  -->
    (modify 2 ^1 chosen)
    (call GOAL indicative <mom>)
    (make adjacent ^to Subject ^is Finite ^mom <mom>))
```

It is possible to have complex entry conditions to systems, and it would be easy enough to use intermediate productions to handle

nested conjunction, as in the case of gates. But recall (from Section 4.2.3) that disjunctive conditions pose a special problem to backward-chaining--this was the reason that independent disjuncts needed to be included as seed features.

Generally, when a disjunctive entry condition is encountered during backward-chaining, it amounts to a choice. The question is: along which path does the best solution lie? Indeed, the wrong choice could lead to a conflict and to backtracking. Faced with disjunctive entry conditions, even goal-directed backward-chaining degenerates to searching explicitly through alternatives.

Fortunately, in many search spaces--including those normally represented by systemic grammars--there is a high degree of interdependence between the various branches of the search path. This can be exploited by the technique of least commitment (Stefik et al., 1983). Least commitment suggests that when a disjunctive entry condition is encountered, the choice should just be left pending since it is very likely that one of the disjuncts will appear in the solution to another goal, thus gratuitously resolving the pending disjunction without choice. In systemic terms, if there are several possible entry conditions to a goal feature, the choice should be delayed. If one of the entry conditions is chosen as part of another backward-chain of reasoning, then there is no longer any problem--the entry conditions of the goal feature are satisfied.

Generally, there will always be the possibility that none of the disjuncts ever appear gratuitously, in which case some form of search must take place. The policy in SLANG-I is for the semantic stratum to provide enough knowledge so that there is no blind search whatsoever. The case where search would be needed is exactly the case of independent disjuncts, and the avoidance of this search is the reason that independent disjuncts are preselected. Thus the resolution of disjunctive entry conditions is guaranteed. For this reason it was decided to simplify SLANG-I by not implementing least commitment at all--disjunctive entry conditions are simply ignored entirely. This does not affect the forward- and backward-chaining

since the disjunction would always be resolved gratuitously by a preselected feature or by another chain of reasoning.

((((a b]-) c =}-) -[) sample)

would be translated into

```
(p sample
  (sub-judice <mom>)
  (goal sample <mom>)
-->
  (modify 2 ^1 chosen)
  (call GOAL c <mom>))
```

assuming that either a or b will be preselected or chosen when achieving another goal. This is a simplifying assumption made in this implementation. If a more sophisticated problem solver were used, it would explicitly use least commitment and would perform the appropriate checks.

7.2.3. The translation

The top level operator of SNORT (see Appendix D, Section 5), takes as an argument a list of feature descriptions as described above. It simply loops through this list, translating each feature (skipping over the system labels--identified by being in capitals) into OPS5 productions. For each feature, the operator "f-p" (feature to production) is called with the following arguments:

a) the feature name,

b) a flag indicating whether or not the feature is "in a system" (i.e. a term in a system, or an entry condition to a feature "in a system"--this determines whether the feature is written as a forward- or backward-chaining production),

c) the entry conditions to the system,*

---

* Note that the entry conditions of a system will not be specified in the same feature description as the feature name if a system label is used. In this case SNORT simply finds the description for the system label, and uses the entry conditions there.

d) a list of the realization rules.

F-p simply builds a production following one of the two templates (depending on the "in-a-system" flag). There are two important operators that are called by "f-p": "decode," which decodes the entry condition description; and "unformat," which translates the systemic format realization rules into realization statements in the OPS5 attribute-value notation.

Decode takes the nested entry condition descriptions described above (including =}-, -[, -<> etc.) and returns a list of conjuncts where any sublists are disjuncts (there is no further nesting). Thus "f-p" can easily take this and write each element of the list as a condition in the case of gates, or ignore the disjuncts in the case of systems. Thus it is "decode" that is responsible for building the intermediate productions for gates with nested conjunction in their entry conditions. This is done very easily, however, by simply calling "f-p" with trivial arguments (where the name of the intermediate production is made by concatenating the original feature name with a unique symbol returned by gensym). Since "f-p" and "decode" are mutually recursive in this way, arbitrary levels of nesting can be handled.

The other important operator called by "f-p" is "unformat." In most cases the unformatting of realization rules is trivial. For instance if the argument is "(F1 / F2)", "unformat" returns "(make conflate ^fun F1 ^with F2 ^mom <mom>)". The only difficult case is preselection. The rule "(Goal<Deictic<Head : !singular)", is unformatted to "(preselect ^feature !singular ^for Goal Deictic Head ^mom <mom>)" where "Goal Deictic Head" is a path describing the place in structure to which the feature applies.

Aside from a few subtleties, in particular the creation of

For instance, the entry condition of masculine in its feature description is GENDER. So the description of GENDER is found, and the entry conditions (third and singular) are used in the production for masculine.

intermediate productions, SNORT is a very simple program. The top-level operator takes a second argument, a file name, and the resulting OPS5 productions are pretty-printed to this file. Once the system network has been SNORTed, the productions can be loaded into OPS5 any number of times. SNORT only needs to be used again if changes are made to the network.

7.3. SLANG-I

The first implementation of the "Systemic Linguistic Approach to Natural-language Generation" generates text using the the OPS5 productions output by SNORT. Aside from SNORT, the rest of the system can be divided into two parts: first, the linguistic productions that interpret the realization statements and build the structure of the text; second, the non-linguistic productions and operators responsible for the low-level workings of the system. These will be discussed in the following two sections.

7.3.1. Realization productions

This section describes the OPS5 realization productions, which interpret the realization statements put in working memory during the processing of the grammar. These productions are responsible for building a linguistic structure for the text being generated.

OPS5 grammar productions use the "make" command to put elements in working memory. The first three productions look for realization statements in working memory that involve functions which do not yet have function descriptions. In each case a function description is created with the "make" command, and a unique hub is created using the built-in OPS5 function "genatom." Remember when reading the OPS5 productions that only those fields relevant to the pattern matching process will appear in the LHS of the production, and that the attribute-value pairs may appear in any order.

```
;;;IF   there is a conflate lexify or expand statement
;;;     whose first argument does not have a hub,
;;;THEN create a unique hub for it.
(p new-hub::first-arg
  (<< conflate lexify expand >> ^fun <f> ^mom <m>)
 -(hub ^of <f> ^mom <m>)
-->
  (make hub ^of <f> ^is (genatom) ^mom <m>))


;;;IF   there is a conflate statement whose second
;;;     argument does not have a hub,
;;;THEN create a unique hub for it.
(p new-hub::second-arg:conflate
  (conflate ^with <f> ^mom <m>)
 -(hub ^of <f> ^mom <m>)
-->
  (make hub ^of <f> ^is (genatom) ^mom <m>))


;;;IF   there is an expand statement whose second
;;;     argument does not have a hub,
;;;THEN create a unique hub for it.
(p new-hub::second-arg:expand
  (expand ^into <f> ^mom <m>)
 -(hub ^of <f> ^mom <m>)
-->
  (make hub ^of <f> ^is (genatom) ^mom <m>))
```

For instance, suppose a conflation statement is put in working memory during the processing of the grammar:

(conflate ^fun Subject ^with Agent ^mom g00007)

Furthermore, suppose neither of these functions appears in any of the realization statements processed at this particular point in structure--and thus they do not yet have hubs assigned to them. Note that the first condition of the first production is matched by this conflation statement (conflate ^fun Subject ^mom g00007), and ex hypothesi the second condition is met since there is no working memory element (hub ^of Subject ^mom g00007). Thus the first production can fire resulting in (for instance)

(hub ^of Subject ^is g00010 ^mom g00007)

being added to working memory. Similarly, the second production

also matches (conflate ^with Agent ^mom g00007), and since there is no (hub ^of Agent ^mom g00007) this production can also fire adding (for instance)

(hub ^of Agent ^is g00011 ^mom g00007)

to working memory. For the actual conflation see the description below.

Hubs are created in a similar manner for functions appearing in lexify and expand statements. Preselection and adjacency are both special cases. Recall that preselection statements can contain an arbitrary number of functions in the paths. Thus it is easiest to write a special production for preselection that calls LISP to loop through the list of functions in the path (see below). Adjacency is special because it does not require changes in the function descriptions (note that there is no ^adjacent attribute for the functions). In fact the adjacency statements put in working memory by the grammar are simply used as they stand.

The OPS5 productions that actually interpret the realization rules of the grammar can now be described. These productions will not fire until hubs have been created for all functions involved in the productions above. The first production is for expansion:

```
;;;IF   an expand statement is encountered, expanding a
;;;     function into subfunctions
;;;THEN put the hub of the expanded function as the ^super
;;;     of the hub of the subfunction.
(p expand
 {(expand ^fun <f> ^into <subf> ^mom <m>)        <expand>}
  (hub ^of <f> ^is <h> ^mom <m>)
 {(hub ^of <subf> ^mom <m>)                  <hub-of-subf>}
-->
  (modify <hub-of-subf> ^super <h>)
  (remove <expand>))
```

This is a very simple production that simply inserts a value for ^super into the subfunction's description. For instance, if (expand ^fun Mood ^into Subject ^mom g00012) and (hub ^of Subject ^super nil ^mom g00012) and (hub ^of Mood ^is g00004 ^mom g00012) are all in working memory, then the description of Subject will be modified to

(hub ^of Subject ^super g00004 ^mom g00012).

The "conflate" production itself simply recognizes the need for a conflation and then sets a task that will be performed by another set of productions.

```
;;;IF   a conflate statement is encountered and the functions
;;;     have different hubs,
;;;THEN substitute the hub of the first for all instances
;;;     of the hub of the second by setting the task
;;;     change-hub.
(p conflate
 {(conflate ^fun <f1> ^with <f2> ^mom <m>)      <conflate>}
  (hub ^of <f1> ^is <h1> ^mom <m>)
  (hub ^of <f2> ^is { <h2> <> <h1> } ^mom <m>)
-->
  (remove <conflate>)
  (make task change-hub <h2> to <h1>))
```

For instance, if the conflation statement (conflate ^fun Subject ^with Agent ^mom g00007) and (hub ^of Subject ^is g00010 ^mom g00007) and (hub ^of Agent ^is g00011 ^mom g00007) are all in working memory, then the element (task change-hub g00011 to g00010) is put in working memory, activating the following set of productions:

```
;;;IF   any functions are associated with the old hub,
;;;THEN associate them with the new one instead.
(p change-hub::functions
  (task change-hub <old> to <new>)
 {(hub ^is <old>)                              <hub>}
-->
  (modify <hub> ^is <new>))
```

The above production is useful in cases where there are several functions associated with the to-be-replaced hub (as a result of previous conflations).

```
;;;IF   any attribute-value statements have the old hub
;;;     in the ^mom field,
;;;THEN change it to the new one.
(p change-hub::mothers:value-attribute
  (task change-hub <old> to <new>)
 {(<< hub adjacent >> ^mom <old>)       <hub/adjacent>}
-->
  (modify <hub/adjacent> ^mom <new>))
```

```
;;;IF   any vector statements have the old hub in the
;;;     last (mom) field,
;;;THEN change it to the new one.
(p change-hub::mothers:vector
  (task change-hub <old> to <new>)
 {(<< chosen goal >> {} <old>)                    <vector>}
-->
  (modify <vector> ^3 <new>))


;;;IF   any ^super fields have the old hub
;;;THEN change it to the new one.
(p change-hub::super
  (task change-hub <old> to <new>)
 {(hub ^super <old>)                              <hub>}
-->
  (modify <hub> ^super <new>))
```

The task "change-hub" has the effect of simply substituting the new hub symbol for the old one throughout working memory.

The production for lexification, like that for expansion, simply adds a value to a field in the function description--in this case the ^lex field.

```
;;;IF   a lexify statement is encountered,
;;;THEN associate the lexical item with the function's hub.
(p lexify
 {(lexify ^fun <f> ^as <lex> ^mom <m>)            <lexify>}
 {(hub ^of <f> ^is <h> ^mom <m> ^output? <> yes) <hub>}
-->
  (modify <hub> ^lex <lex> ^output? no)
  (remove <lexify>))
```

The only moderately difficult realization is preselection. Here the difficulty is not because preselection itself is especially complex, but merely because the paths in preselection involve a variable number of functions, and therefore are not easily handled directly by the OPS5 pattern matching facility.

However, OPS5 allows LISP operators to be called from the RHS of productions.

```
;;;IF   a preselection statement is encountered
;;;THEN pass the whole thing to a lisp operator that
;;;      handles this.
;;; [This is necessary because the number of arguments
;;; is arbitrary.]
(p preselect
 {(preselect ^feature <feature> ^mom <m>)     <preselect>}
-->
  (call PRESELECT (substr <preselect> 1 inf))
  (remove <preselect>))
```

The LISP operator PRESELECT (see Appendix D, Section 4) receives a feature and a path (list of functions) as arguments. It loops through the list of functions where each function appears in the ^mom field of the next function in the list (if necessary, a new hub and function description are created). Finally, a goal statement for the feature is put in working memory, with the mother corresponding to the last element in the list.

Suppose a preselection statement is put in working memory during the processing of the grammar: (preselect ^feature !feminine ^for Agent Deictic Head ^mom g00006). Suppose none of these functions have hubs already. PRESELECT is called and begins looping through the list (Agent Deictic Head). The Agent in this case is a function at the current place in structure, so an element such as (hub ^of Agent ^is h00004 ^mom g00006) is created with the help of gensym. The further elements (hub ^of Deictic ^is h00005 ^mom h00004) and (hub ^of Head ^is h00006 ^mom h00005) are then created. Thus the path as a whole points to the place in structure occupied by the hub h00006. Accordingly, an element (goal !feminine h00006) is placed in working memory to actually perform the preselection.

7.3.2. The support system

The other group of productions to be described are those responsible for the low-level workings of the system. These productions are concerned with two related aspects of the system. The productions are responsible for keeping track of what parts of the text have been output. Also, most of the productions are used to coordinate the system so that the text is generated from left to

right.

The first production can be found in almost any OPS5 program:

```
;;;IF   a task is no longer appropriate
;;;     (it matches no other productions),
;;;THEN delete it.
(p remove-task
 {(task)                                  <task>}
-->
  (remove <task>))
```

OPS5 will match statements with more specific (more vector places specified) conditions first; since this condition is the least specific possible, it is guaranteed to match last, that is when the task has already been done (see Waltzman, 1983, p. 29).

The next production is used to mark functions conflated with an output function as also being output.

```
;;;IF   a hub has been output,
;;;THEN make sure all the wm elements for that
;;;     hub are marked accordingly.
(p spread-output
  (hub ^is <h> ^output? yes)
 {(hub ^is <h> ^output? <> yes)          <out-of-date>}
-->
  (modify <out-of-date> ^output? yes))
```

The remaining productions involve the idea of a node being sub-judice, or under consideration. The idea is that nodes in the structure are marked as being sub-judice progressively from left to right as more of the text is actually output. Recall that the backward-chaining productions in the grammar require the place in structure (the mother node) to be sub-judice before they can fire. Thus when the nominal group realizing the leftmost constituent in a clause (say the Topical) is sub-judice, productions for this nominal group will fire first. The sub-judice "marker" is only moved to the right once the constituents to the left have actually been output. Of course it is slightly more complicated than this because the structure is a graph resembling a tree, not a list. Since OPS5 gives the most recent working-memory elements priority, if the structure graph is created depth-first, the deepest element has the

highest priority, but the ancestors are also sub-judice. So going back to the previous example, the productions will fire for the nominal group if possible, but if there have not been enough preselections, the generation of the Topical will eventually get stuck. The next most recent node (the clause) then has some more of its productions fired until a preselection or conflation (e.g. Topical / Subject) gives the nominal-group productions some more information to work with. Thus in general, the sub-judice constraint forces the generation to proceed in an in-order depth-first manner.

It may appear that there is a possibility of the generation getting "deadlocked" if a preselection is required from a non-sub-judice node. It becomes clear that deadlock can not occur, however, when it is considered that the only source of input to the generation at a particular node is preselection, and preselection can only come from an ancestor in the constituent tree, and all ancestors of a sub-judice node are required to be sub-judice themselves. Therefore, because preselection is directional, deadlock cannot occur.

The following production is a result of the possibility of a hub having more than one supernode. For instance, if the Topical is conflated with the Subject (a common occurrence), then the common hub will be a subnode of both the Theme (the supernode of Topical) and Mood (the supernode of the Subject). This production ensures that if the Subject is sub-judice by virtue of being conflated with the Topical, then the Mood should also be sub-judice.

```
;;;IF   a node is under-consideration (sub-judice),
;;;THEN make sure any supernodes are also
;;;      under consideration.
(p fill-scope::super
  (sub-judice <sj>)
  (hub ^is <sj> ^super { <super> <> nil })
 -(sub-judice <super>)
-->
  (make sub-judice <super>))
```

The following production actually outputs a lexical item. If a hub is sub-judice and it has a lexical item associated with it (see lexify above) then simply write out the lexical item and mark the hub as output.

```
;;;IF   a lexical item has been associated with a hub,
;;; and that hub is sub-judice,
;;;THEN output that lexical item
;;; and mark the hub as output.
(p output
 {(hub ^is <h> ^lex { <l> <> nil } ^output? no)    <hub>}
  (sub-judice <h>)
-->
  (write <l> (crlf))
  (modify <hub> ^output? yes))
```

The final set of productions is responsible for the flow of control through the structure network. The first of these productions moves the sub-judice marker down one level of unit nesting.

```
;;;IF   a node is sub-judice and not output,
;;;THEN declare its leftmost child (if there is one)
;;;     sub-judice.
(p move::down:#
  (sub-judice <sj>)
  (adjacent ^to # ^is <f> ^mom <sj>)
  (hub ^of <f> ^is <h> ^output? <> yes ^mom <sj>)
 -(hub ^is <sj> ^output? yes)
-->
  (make sub-judice <h>))
```

The second production moves the sub-judice marker down to the leftmost subnode of a node that is sub-judice.

```
;;;IF   a node is sub-judice and not output,
;;;THEN declare its leftmost subnode (if there is one)
;;;     sub-judice.
(p move::down:%
  (sub-judice <sj>)
  (adjacent ^to % ^is <f> ^mom <m>)
  (hub ^of <f> ^is <h> ^super <sj> ^output? <> yes ^mom <m>)
 -(hub ^is <sj> ^output? yes)
-->
  (make sub-judice <h>))


;;;IF   a hub has just been output
;;;THEN declare the node adjacent to it (if there is one)
;;;     sub-judice.
(p move::across
  (hub ^of <f> ^output? yes ^mom <m>)
  (adjacent ^to <f> ^is <f1> ^mom <m>)
  (hub ^of <f1> ^is <h1> ^output? <> yes ^mom <m>)
-->
  (make sub-judice <h1>))

;;;IF   the rightmost child of a node has just been output,
;;;THEN declare the node output
(p move::up:#
  (adjacent ^to <f> ^is # ^mom <m>)
  (hub ^of <f> ^output? yes ^mom <m>)
 {(hub ^is <m> ^output? <> yes)                    <mother>}
-->
  (modify <mother> ^output? yes))
```

```
;;;IF   the rightmost subnode has just been output,
;;;THEN declare the supernode output.
(p move::up:%
  (adjacent ^to <f> ^is % ^mom <m>)
  (hub ^of <f> ^output? yes ^super <super> ^mom <m>)
 {(hub ^is <super> ^output? <> yes)          <supernode>}
-->
  (modify <supernode> ^output? yes))
```

Finally, there are the two LISP operators, GOAL and ASSERT (see Appendix D, Section 4), mentioned earlier. Unlike PRESELECT, which was written in LISP by necessity, GOAL and ASSERT have been written in LISP for reasons of efficiency. Almost every grammar production calls one of these two operators, so they greatly affect the efficiency of the system. (call GOAL feature <mom>) and (call ASSERT chosen feature <mom>) are equivalent to (make goal feature <mom>) and (make chosen feature <mom>) respectively, except that in each case the external routines check to make sure that feature has not already been chosen. This turns out to be faster than putting a -(chosen feature <mom>) condition in every grammar production.

The reason these external routines are so fast is that they take advantage of the hashing mechanism used by OPS5. Working memory elements are hashed on their first field. The elements that have the form (chosen ...) are stored in a list separate from the rest of working memory. A quick look at the source code of OPS5 reveals that this list can be retrieved by (get 'chosen 'wmpart*). Thus the feature check can be made by scanning a fraction of working memory.

7.4. Limitations of the current implementation

Some limitations of SLANG-I resulted from shortcuts taken to reduce the development time, complexity, and running time of the system. For instance, no checks are made to make sure exactly one feature is chosen from a system whose entry conditions are satisfied. No checks are made that preselected features actually exist. It is assumed that the networks form a coherent whole and that the input to the system will be reasonable. These checks would be easy

to implement, but were excluded for reasons of simplicity and execution speed.

Several other limitations are the result of one major shortcut, viz. that what is essentially the OPS5 inference engine has been used as the "problem solver." Ideally a more sophisticated problem solver would perform forward- and backward-chaining. It would also have built-in mechanisms to handle disjunctive conditions (e.g. least commitment--see Stefik et al., 1983) and would eliminate the need for GOAL and ASSERT. The results would be the same, but the mechanism would be less ad hoc.

7.5. Alternative implementations

There is no doubt that SLANG-I's straight OPS5 production rule implementation was only one of many possibilities. This section will suggest that other production systems and even object-oriented languages could, and perhaps should, be used in future implementations.

7.5.1. Other production systems

The advantages of essentially using the actual OPS5 mechanism as the problem solver for this initial implementation were its simplicity and speed. In practice, however, SLANG will be implemented within an expert system where the problem solving is done by a much more sophisticated inference engine. This problem solver may, for instance, have the ability to reason about the RHS of the production rules instead of simply trying to match the LHS. This may, for instance, enable the problem solver to "reason from first principles" by reasoning with the functional relationships described in the realization rules at the grammatical stratum in cases where no suitable compiled knowledge is available.

The point is that even with a representation almost identical to that used in SLANG-I, problem-solving techniques that are much more sophisticated than those built into OPS5 can be applied to text generation by a more sophisticated problem solver.

It must be considered, however, that speed will always be an *important issue in practical text-generation work*. Lengthy reasoning about language can only be tolerated in some rare circumstances. Thus, even though reasoning from first principles, as one example, is necessary to round-out SLANG's capabilities, the high-speed compiled approach demonstrated by SLANG-I must remain the dominant form of reasoning.

If a production system is used to implement SLANG then, it must have the ability to do simple forward- and backward-chaining very quickly. It should also have the flexibility to allow access to all parts of the production (including the effects) and efficiently implement backtracking and the techniques to avoid it (e.g. least commitment, Stefik et al., 1983).

7.5.2. Inheritance hierarchies

One interesting possibility is that of representing system networks as inheritance hierarchies. This could be done, for instance, using a semantic network notation, or an object-oriented notation. Clauses, for instance, could be represented as a class of objects, finite clauses as a subclass of these, indicative clauses as a subclass of these, declarative clauses as a subclass of these and so on (note the use of of the term "class" corresponds exactly to the systemic term--see Section 3.3.1). Declarative clauses thus inherit the properties (represented by realization rules) of indicative clauses, finite clauses and clauses. In a sense, the backward-chaining of SLANG-I implements the mechanism of inheritance, but it may be more elegant and efficient if the problem solver performs inheritance as a primitive task. Of course the complex relationships found in system networks (including multiple inheritance) must be able to be represented in whatever notation is chosen. It may be that a hybrid representation (where, for instance, an inheritance hierarchy represents the systems and productions represent the gates) is best.

Indeed, returning to the fundamental relationship described in

Chapter 4, SLANG should be able to be implemented using many of the tools and techniques developed to "construct [a] solution selectively and efficiently from a space of alternatives" (Hayes-Roth et al., 1983).

7.6. Summary

This chapter has discussed some of the implementation issues of SLANG and the implementation of a prototype system. This implementation uses an OPS5 representation for the system networks. The networks are translated from a LISP-based system network notation into the OPS5 productions. Gates are translated into forward-chaining productions, and other features are translated into backward-chaining productions. The realization rules, which appear as effects in the productions, are put in working memory as they are encountered in the form of OPS5 realization statements. Other productions, which are not part of the grammar but were written in OPS5 for convenience, then interpret these realization statements and build the linguistic structure of the text. Still other OPS5 productions perform the low-level behind-the-scenes work including writing out the text as it is generated.

Several shortcuts have been taken to reduce the size and complexity of the initial implementation. These included the omission of error checks, and the use of a slight abstraction of the OPS5 inference engine as the "problem solver" instead of implementing something more substantial. Finally, a straight production system implementation is only one possibility. Other problem-solving representations, in particular inheritance hierarchies--perhaps in conjunction with production rules--may work as well or better.

8. Comparison with Other Work

This chapter will compare and contrast the text-generation method described in the previous chapters with other recent work in the field. This will not include the large body of research done recently on discourse planning, but only work concerned with realizing these plans. The most notable exclusion on these grounds is McKeown's TEXT (McKeown, 1982, 1983) which sets some discourse related goals then does the actual text generation using unguided search and backtracking (see Appelt, 1983, p. 599).

A look at recent systems reveals that there are currently two general approaches to text generation: the "grammar-driven" approach and the "goal-driven" approach. Both of these will be outlined, including their major practitioners, and the advantages that are offered.

Next, the systems that try to combine these two approaches will be considered. It will be shown that SLANG successfully achieves this, capturing the advantages of both the grammar-driven and goal-driven approaches.

8.1. The grammar-driven approach

Several of the major text-generation projects are "grammar-driven." This term will be used to refer to those systems that traverse an explicit linguistic grammar. Since the flow of control is directed by the grammar traversal, the logical structure of the system reflects the structure of the grammar.

The original grammar-driven systems would simply traverse the grammar, typically an ATN, backtracking where necessary. More recent grammar-driven systems avoid backing up by doing an analysis at choice points to make sure the right decision is made the first time. This analysis often involves considering semantic and pragmatic issues that, for reasons of modularity, should not be directly accessible to the grammar. Therefore an interface mechanism of some kind is provided through which the higher level guidance may be obtained. Two grammar-driven systems will now be surveyed.

8.1.1. PROTEUS

PROTEUS (Davey, 1978) is a program for annotating games of noughts and crosses (tic-tac-toe). It produces fluent text in this limited domain, with particular emphasis on referring expressions and ellipsis. PROTEUS traverses an explicit systemic grammar in order of delicacy (from left to right). Some features will already have been chosen when the grammar is entered, but most decisions at choice points are made by "specialists." Some specialists consider the text that has been produced so far, and the "semantic specialists" consult the non-linguistic domain knowledge for guidance. The program either plays a game itself or accepts the moves of a game as input. In either case the game annotation begins with a transcript of a valid set of moves.

An example from (ibid., p. 17):

The following commentary was given on the moves shown:

'The game began with my taking a corner, and you took an adjacent one.	_ | _ | _ P | _ | A
I threatened you by taking the corner adjacent to the one that you had just taken, but you blocked my diagonal and threatened me.	_ | _ | P P | A | _ _ | _ | A
I blocked yours and forked you.	P | _ | P P | A | _ _ | _ | A
Although you blocked one of my edges and threatened me, I won by completing the other.'	P | P | P A | A | _ P | _ | A

The general procedure is to choose some number of consecutive moves to be described in a sentence, to generate this, and to repeat. First the tactical significance of the moves is determined. Then, depending on this, descriptions of the appropriate moves are conjoined to form a sentence of not more than three main clauses.

The actual sentences are formed by traversing system networks. When faced with a choice, PROTEUS calls upon a "specialist

procedure" to make a decision based on the "syntactic and semantic context." These specialists may actually generate bits of text as part of the decision process to see, for instance, if a satisfactory modifier or qualifier can be constructed. Davey claims that

> It will therefore be obvious that the program's operations cannot be categorized as working 'top-down'. It does not invariably construct an item by determining its feature-set, thence determining constituent-structure, and finally building each constituent. Instead, syntax and semantics are woven together and dependent on each other, either one being able to take control as the situation demands. (ibid., p. 120-121).

When traversing the network, PROTEUS uses defaults to save some work. For instance, unless told otherwise it assumes clauses are independent, indicative, declarative, and past. This would not be useful except in the restricted register in which PROTEUS operates.

One interesting aspect of the semantic specialists is that they may choose several features at a time as opposed to the "one feature, one specialist" approach adopted in Nigel (see below). For example, if a semantic specialist decides a relative clause is necessary, it preselects (not a term used by Davey) clause, dependent, finite, and relative. The semantic specialists may also decide that a special time adverb or aspect is required, and again it will preselect the necessary clause features.

PROTEUS has similar specialists to help construct referring expressions. These can be quite complex: "the corner common to the edge opposite the square X had just taken and the one opposite the square O had just taken" (ibid., p. 144).

In summary, PROTEUS produced impressive results in the limited domain but relied on ad hoc procedures and "specialists" to a large degree. Despite the shortcomings of the implementation, PROTEUS became a major influence in text-generation research since essentially the same approach was adopted for the Nigel system.

8.1.2. Nigel

Nigel (Mann et al., 1983; Mann, 1985) is a general purpose text-generation system very similar in design to PROTEUS. It too is built around an explicit systemic grammar, and the grammar itself has been the focus of most of the work. As in PROTEUS, the grammar is traversed in order of delicacy, from left to right. The decisions at choice points (systems) are handled by "choosers," that are organized and documented much better that Davey's specialists. There is also a well-defined interface between the choosers and the "environment." The "environment" knows about the text plan and goals of the speaker, and has access to the non-linguistic domain knowledge. The choosers base their decision on the answers received in response to specific questions posed to the "environment."

As an example of how the grammar is traversed, consider the informal description of how the mood of a clause is chosen (Mann et al., 1983, p. 41). The grammar finds itself in a system where the choice is between indicative and imperative. The chooser asks the environment, "Is the illocutionary point of the surface level speech act ... a command, i.e. a request of an action by the hearer?" The environment then answers, "It's not intended to command." So the chooser chooses indicative. The grammar, having just passed through one system, now finds itself at another labelled "IndicativeType" where the choice is between declarative and interrogative. The chooser for this system asks the environment, "Is the illocutionary point of the surface level speech act ... to state?" And the environment answers, "Yes, it's intended to state." The chooser therefore chooses declarative. Any realization rules attached to the features are processed as soon as they are chosen. This same "question and answer" procedure is repeated for every system encountered during the traversal. The clause network is traversed again for any embedded clauses, the nominal-group network is traversed for any nominal groups required and so on. Nigel will be discussed further in Section 8.4 below.

8.1.3. Advantages of grammar-driven systems

The grammar-driven approach, of which PROTEUS and Nigel are examples, has several important advantages. The grammars may be represented in a linguistic formalism since no processing information needs to be included. This is advantageous because it means that grammar can be judged, modified, understood etc., independently of the rest of the system (Appelt, 1982). In addition, the logical structure of the system is explicit in the grammar. This makes the operation of the system easier to understand.

In general, these advantages are conducive to research projects, like PROTEUS and Nigel, that are primarily concerned with linguistic issues rather than with processing.

8.2. The goal-driven approach

On the other side of the text-generation coin are systems that generate text by goal-directed problem solving. The control in these systems typically rests with some form of text-planner; the flow of control is driven by the goals the planner is trying to achieve. It is the mechanism of this planner and the interface with the grammatical component which is of interest in these systems. The grammar itself is relegated to an obscure if not invisible role. They tend to regard the form of the linguistic component (LC) as being subsidiary to, or even dependent on, the goal-directed problem solving.

> [W]e believe (and it is here that we part company with researchers such as Mann and Matthiessen [1983] whose aims we otherwise share) that the demands placed on the LC by the need to work efficiently from a plan have overriding implications for the LC's architecture (McDonald et al., 1985, p. 800).

Note that a new term--"goal-driven"--has been coined here to make the distinction between systems which merely take goals into consideration, and systems whose control structure is goal-oriented. Consider the choosers in Nigel. The choices that are made often depend on consideration of the goals of the speaker (Mann et al.,

1983). Clearly in some sense Nigel is therefore directed by these goals, but the choosers are invoked in the first place because of their position in the grammar, not because a particular goal emerged.

Two examples of goal-driven text generators are Appelt's KAMP and the work involving McDonald's MUMBLE.

8.2.1. KAMP

KAMP (Appelt, 1982) is a planner that integrates linguistic and other types of actions to achieve communicative goals. The important issues in KAMP are this integration, and the ability to reason about the knowledge and intentions of other discourse participants.

Consider a situation (from ibid., p. 2) where there are two agents, A and B, working in a shop where there are several objects on a table. Suppose agent A knows agent B wants to perform a particular task. Suppose agent A wants to help agent B by telling him to use a particular tool with which agent B is unfamiliar. Agent A may, for instance, point to one of the objects and say "Use the wheel-puller to remove the flywheel." In this case the speaker has combined a non-linguistic act (pointing) and a linguistic act to achieve two goals: first to communicate which tool to use, and second to communicate the name of the tool for future use.

Suppose further that, in a similar situation, the speaker had his hands full and there was no satisfactory verbal description of the wheelpuller. The speaker would then have to plan to put down what he was carrying and again point to the tool while giving his advice. Thus there is a potential for complex interactions between linguistic and non-linguistic acts that are integrated. These interactions can be resolved, however, by an AI problem solver. KAMP is the problem solver Appelt constructed to explore and illustrate these ideas.

In the last example it was important that the speaker knew that the hearer didn't know the name of the necessary tool--otherwise the

pointing action is redundant. Similarly, the entire episode would have been unnecessary if agent A had known that agent B already knew what tool to use. It is clear that in communication of this sort, reasoning about the knowledge of the agents involved is required. Appelt illustrates this kind of reasoning with an example (ibid., p.83):

> Consider the following problem:
>
> > A robot named Rob and a man named John are in a room that is adjacent to a hallway containing a calendar. Both Rob and John are capable of moving, reading calendars, and talking to each other, and they each know that everyone is capable of performing these actions. They both know they are in the room, and they both know where the hallway is. Neither Rob nor John knows what date it is. Suppose further that John wants to know what day it is, and Rob knows he does. Furthermore, Rob is helpful and wants to do what he can to ensure that John achieves his goal. We would like to see KAMP devise a plan, perhaps involving actions by both Rob and John, that will result in John knowing what day it is.
>
> > We would like to see Rob devise a plan that consists of a choice between two alternatives. First, if John could find out where the calendar is, he could go to the calendar and read it, and in the resulting state would know the date. So, Rob might tell John where the calendar is, reasoning that this information is sufficient for John to form and execute a plan that would achieve his goal. The second alternative is for Rob to move into the hall and read the calendar himself, move back into the room, and tell John the date.

KAMP solves the problem by doing hierarchical planning (Sacerdoti, 1975) with predicates and instantiations such as KnowsWhatIs(Rob, date), and actions such as Do(Rob, Move(loc(Rob), Loc(cal1))) and Do(Rob, Inform(John, Date=D)).

Thus KAMP can solve this type of problem by goal-directed reasoning about the mental states of agents including itself, and about the interactions between the linguistic and non-linguistic acts involved. But unfortunately, the grammar is built into the planner ad hoc and is thus not easily accessible or observable (Appelt, 1983).

8.2.2. MUMBLE

The other major goal-driven work is that of McDonald (1980, 1983a, 1983b and McDonald et al., 1985). This, like KAMP, is primarily concerned with the achievement of goals and the organization of language generation, not grammatical issues.

McDonald labels his control structure as "description directed control." The description is of the final text, but at a very high level of abstraction. This description can be regarded as a special notation for specifying sets of goals.

The general organization of McDonald's most recent work (McDonald et al., 1985) is that of a pipeline performing four operations concurrently. The first step is "planning," using specific script-like knowledge for specific linguistic registers. In the example domain, legal discourse, these scripts are "Describe-legal-case," "Describe-a-party-to-a-case," "Describe-corporate-party," etc. Additional register information is included in the plan labelled as "perspectives," such as "establish-relation-of-speaker," and "misappropriation-script."

The second step is "attachment." The parts of the plan are attached to a phrase-structure representation according to various grammatical constraints and some "stylistic rules" indicating the preferred length and complexity of sentences for a particular register.

The third step in the pipeline is "realization." Here the high-level representation attached to the phrase-structure tree is realized as a phrase-structure subtree. The process annotates the nodes with functional labels and morphological information to help constrain further processing.

The fourth step in the pipeline is the "phrase-structure execution." At this step the phrase-structure representation built by the last step is traversed, recursively traversing subtrees and outputting lexical items attached to leaf nodes.

The interest here is in what information gets passed when between the various components. Notice that the semantic knowledge here, unlike that of the grammar-driven systems, can be organized by semantic criteria. For instance it is explicitly stated that "Describe-corporate-party" is a sub-type of "Describe-a-party-to-a-case" (McDonald et al., 1985). The grammar in MUMBLE, like that of KAMP, is built into the workings of the program and is not presented as being a major issue.

8.2.3. Advantages of the goal-driven approach

The goal-driven approach also has some important advantages. One particularly important characteristic of the goal-driven approach is that the semantic knowledge can be organized according to semantic criteria instead of being tied to the grammar. When using the grammar-driven approach, the semantic knowledge responsible for interfacing the extralinguistic inference with the grammar must be attached to choice points in the grammar. No such constraints are imposed on goal-driven systems.

Another advantage is that, as KAMP demonstrated, non-linguistic acts can be integrated with linguistic acts to achieve goals. This is not possible in a strictly grammar driven approach because the grammar contains no extralinguistic knowledge.

More abstractly, the goal-driven approach seems to be more consistent with the aims of the computational paradigm than the grammar-driven approach. This is not surprising since the emphasis in this paradigm is on processing.* The computational paradigm (see Section 5.3) tries to apply general problem-solving techniques to processing language. One of the most powerful problem-solving techniques is goal-directed search (see 2.1.4) which, applied to language generation, is the goal-driven approach.

---

* Note, however, that McDonald takes a generative stance (see the introduction to McDonald, 1983b).

8.3. Combining the approaches

Both the grammar-driven and goal-driven approaches have advantages. Ideally the explicit grammar and logical structure of the grammar-driven approach could be combined with the goal-oriented problem-solving approach reflecting communicative goals of the speaker.

8.3.1. TELEGRAM

TELEGRAM (Appelt, 1983) was an attempt to combine KAMP's emphasis on planning throughout the generation with an explicit grammar (in this case a functional unification grammar). Although TELEGRAM had a planner *available* throughout the generation, the primary locus of control was in the grammar.

> [T]he TELEGRAM planner will create a high level functional description of the intended utterance. ... At this point, the planner is no longer directly in control of the planning process. The planner invokes the unifier with the above text functional description, and the grammar functional description, and relinquishes control to the unification process.
>
> The unification process follows the [unification algorithm] until there is either an alternative in the grammar that needs to be selected, or some feature in the text FD does not unify with any feature in the grammar FD. (Appelt, 1983, p. 598)

For cases where the unification does in fact fail, the grammar is "annotated" with special signals to tell the unifier to invoke the planner with certain goals. Suppose the unifier is trying to unify a functional description for an NP with the corresponding part of the grammar. Suppose that the unification will not be successful because there is no *referent* feature in the textual functional description to unify with the grammar (ibid., p. 598). The *referent* feature, however, may have an annotation that tells the unifier to invoke the planner to plan the referent. The planner will reason, as KAMP did, about the knowledge of the discourse participants and work toward a functional description suitable for the unification at hand. If no suitable plans can be found using linguistic acts, the

planner can insert non-linguistic acts (e.g. pointing) into the plan as KAMP did.

Although descriptions of TELEGRAM place heavy emphasis on the role of the planner, it really has the design of a grammar-driven system similar to Nigel: "the system 'choosers' of Nigel play a role similar to the annotation on the alternatives in TELEGRAM, and many other parallels can be drawn" (ibid., p. 599). Thus, like Nigel, TELEGRAM appears to have to organize a large part of the semantics by grammatical criteria since the annotations are attached to individual grammatical features. In addition, the process is still driven by the unification algorithm:

> In spite of its advantages, there are some serious problems with unification grammar if it is employed straightforwardly in a language planning system. One of the most serious problems is the inefficiency of the unification algorithm ... A straightforward application of that algorithm is very expensive, consuming an order-of-magnitude more time in the unification process than in the entire planning process leading up to the construction of the text FD. The problem is not simply one of efficiency of implementation. It is inherent in any algorithm that searches alternatives blindly and thereby does work that is exponentially related to the number of alternatives in the grammar. Any solution to the problem must be a conceptual one that minimizes the number of alternatives that ever have to be considered. (ibid., p. 596).

Although TELEGRAM's planner reduces this problem, the unification process is still in control, and in common with Nigel this prevents other problem-solving techniques (like backward-chaining) to be used to reduce even further the number of alternatives considered. Whenever a choice is encountered by the unifier, the problem solver is invoked to make a decision based on the "annotations" attached to the various choices. The point is that the annotations of all the choices have to be considered explicitly.

In summary, TELEGRAM achieves the advantages of a grammar-driven system while maintaining KAMP's ability to integrate linguistic and non-linguistic actions. However, two of the advantages of goal-driven systems have still been sacrificed: the ability to organize the semantics independently of the grammar, and the

computational power and flexibility of having the problem solver in control.

8.3.2. SLANG

As TELEGRAM demonstrated, the difficulty in achieving the advantages of both the grammar-driven and goal-driven approaches lies in interfacing the goal-directed problem solver with a linguistically formalized grammar. The SLANG system does not have this problem because of the conflation described in Chapter 4.

SLANG's grammar, even though it is represented in a linguistic formalism, can be interpreted as problem-solving knowledge by a goal-directed problem solver. The key phenomenon here is that the control dictated by the goals and the control dictated by the grammar are the same--they have been conflated. This makes SLANG both a grammar-driven and a goal-driven method.

8.3.2.1. SLANG as a grammar-driven method

The conflation of grammar and knowledge base in SLANG gives it the desirable characteristics of the grammar-driven systems. Specifically, this means that the grammar can be judged, modified etc. independently of the rest of the system. The logical structure is also explicit in the structure of the grammar, making the method comparatively easy to visualize and understand. Compare the two methods of traversing the same type of grammar: SLANG (see Section 4.3) and Nigel (see 8.1.2). In SLANG, the compiled knowledge in the semantic stratum, through the preselection of seed features, constrains the right-to-left traversal of the grammar. Less delicate systems are no longer explicit choice points in the sense that the feature to be chosen is determined by these constraints. Nigel, without the benefit of these constraints, must explicitly consider many more choice points during its left-to-right traversal.

Also, consider the problem Nigel has with the ordering of choices. When the grammar enters the system to choose the number of an indicative Subject, the chooser asks the environment, "Is the [Subject] inherently multiple, i.e. a set of collection of things, or unitary?" (Mann et al., 1983). The problem is that at this point in the traversal, the Subject may not yet be conflated with another

function for which this information is available. In that case the environment cannot answer the question. Mann suggests that it would be unreasonable to suspend the decision until later because the entire system might eventually get stalled. He therefore suggests that the grammar could be rewritten so that the choices are guaranteed to be made only when the information is known at that point in the traversal (Mann, 1985). This, however, would seem to violate the principles of the grammar-driven approach because the linguistic clarity is being sacrificed for computational reasons.

Note that SLANG does not suffer from this problem because, as demonstrated in Chapter 6, the only explicit choice points are the seed features that are chosen by the semantic stratum.

Despite the differences between SLANG and Nigel, SLANG is still a grammar-driven method. An explicit grammar is still traversed and provides the logical structure of the method--the difference is that features in SLANG are also interpreted as goals.

8.3.2.2. SLANG as a goal-driven system

The conflation of grammar and knowledge base also means that SLANG is a goal-driven system and thus inherits the desirable characteristics of that approach. The control structure is goal-directed backward-chaining starting from communicative goals (in the form of semantic features).

Like other goal-driven systems, SLANG has the ability to organize semantic knowledge by semantic criteria. In fact the semantic stratum as presented in (Halliday, 1978) and as implemented in SLANG is much further advanced--in that it is part of a carefully constructed and substantial theory--than the facilities described in (McDonald et al., 1985).

Since, like other goal-driven systems SLANG's activities are initiated and governed by a general-purpose problem solver, theoretically it has the ability to integrate non-linguistic and linguistic acts. Although this line of research has not been pursued, unlike

other grammar-driven systems SLANG has this potential.

Finally, SLANG inherits the advantage of being able to apply powerful AI techniques to the generation, with all the computational and theoretical benefits this implies.

8.4. Problem-reduction in Nigel and SLANG

For the purpose of clearly illustrating the grammar-driven nature of Nigel, the above (Section 8.1.2) discussion glossed over some important issues. This, together with the fact that Nigel is currently the best-known systemic text-generation system, motivates some further discussion of the differences between Nigel and SLANG.

One of the most important issues that was glossed over is that in the case of features that are preselected by realization rules at the grammatical stratum, Nigel has a process called "path augmentation" that seems to be equivalent to SLANG's backward-chaining.

> Any feature in the system network has one or more paths leading to it, i.e., a set of choices through which it can be reached. As long as there is only one path leading to the feature, it can be preselected, and its path computed through redundancy, so-called path augmentation. In other words, on a unique path only the most delicate feature need be preselected. (Mann et al., 1983, p. 68).

Also note that there are no important differences in how the gates are processed.

The key distinction between Nigel and SLANG is the interface between the semantics and the grammar. Nigel does the grammar-driven traversal discussed earlier and SLANG has the semantic stratum preselect grammatical features. The interesting point here is that both of these approaches can be interpreted as problem-reduction (see Section 6.3).

Nigel can be viewed as having a problem-reduction methodology where the start nodes of the AND/OR graphs are the root features of the system networks. So the "initial problem" to construct a clause, for instance, is reduced to making it either finite OR non-

finite, AND making it either material-process, mental-process, verbal-process OR relational-process, and so on. The problem of making it finite is reduced to making it either indicative OR imperative and so on through the entire system network.

The problem that Nigel faces is that most of the reductions are to OR nodes, since in this formulation any feature that is a term in a system will be represented by an OR node. Thus the problem solver does not know which of the OR nodes should be reduced. The choosers make these decisions after interacting with the environment.

In SLANG, the start nodes are seed features and the AND/OR graphs stretch from them back to the root features. The result of this, the preselection of independent disjuncts, and being able to treat dependent disjuncts as solved, is that there are no OR nodes in SLANG's AND/OR graph. This means that once the initial goals have been set, SLANG can proceed deterministically with no need for interaction with other components and only considering those nodes that will be in the solution graph (see Section 6.3). The compiled knowledge allows SLANG to set seed features as initial problems and do problem reduction against the grain of the systems, avoiding the disjunction. SLANG and Nigel produce the same result, and the same implicit grammatical choices are made by the two methods (i.e. the grammatical search space is the same), but the compiled knowledge in the semantic stratum and the goal-driven processing make SLANG's search more selective and more efficient.

A question which could be asked at this point is: "Intuitively speaking, if both Nigel and SLANG are doing problem-reduction by goal-directed backward-chaining, why is there such a difference?" The answer is that the goals in Nigel's case (the start nodes) are vacuous--saying that the solution must be a clause conveys almost no information to the problem solver. The goals SLANG works from, unmarked-declarative-theme, addressee-subject and so on, provide enough information for the problem solver to find the solution selectively without recourse to outside help. This is a result of the compiled knowledge contained in the semantic stratum exploited by SLANG--the strictly grammar-driven nature of Nigel prevents the grammar traversal from being compiled out.

8.5. Summary

This chapter has briefly reviewed the two major approaches to text generation, the grammar-driven approach and the goal-driven approach, and shown that SLANG in fact fits both descriptions.

The grammar-driven approach has the advantages of using an explicit grammar represented in a linguistic formalism. Since this grammar is simply traversed, the control structure of the system is explicit. The goal-driven approach has the advantages of allowing non-linguistic and linguistic acts to be integrated, allowing the semantics to be organized by semantic criteria, and using powerful AI techniques throughout the generation process.

The difficulty of interfacing a goal-directed problem solver with a linguistic grammar is overcome by the conflation of grammar and knowledge base. As a result, SLANG benefits from the advantages of both the grammar-driven and goal-driven approaches to text generation.

9. Conclusions

This final chapter consists of four parts. First, the main points from the previous chapters will be summarized, giving a condensed description of the work done on the SLANG approach to text generation. Second, the problems that may impede progress on SLANG will be examined. Third, some ideas for future research will be explored. Fourth, the concluding remarks will include an evaluation of SLANG and the current progress, and the prospects for the future.

9.1. Summary

9.1.1. The problem

One problem that has persistently occupied and bedevilled text-generation research is how to interface higher-level reasoning with an explicit grammar written in an established linguistic formalism. This problem is central to text generation because of the computational and linguistic requirements of the task.

Text generation involves an enormous, complex search space, yet must be performed quickly if it is to be effective. These characteristics suggest that text generation requires the powerful knowledge-based computational methods--such as forward-chaining and goal-directed backward-chaining--developed in AI over the past fifteen years.

Text generation also has important linguistic requirements. Specifically, an explicit grammar that is represented in an established linguistic formalism is required. This enables direct input from linguists and the linguistic literature. It also allows the grammar to be understood, judged, modified and so on, independently of the computational concerns (Appelt, 1982). Finally, assuming that the processing is guided by the explicit grammar, the grammar can provide a useful display of the logical structure of the text-generation process.

The problem of interfacing the AI problem-solving techniques with the linguistic formalism arises because of the apparent incompatibility of the representations involved. The computational representation on the one hand has been developed with issues such as selectivity and efficiency in mind. The linguistic representations on the other hand have been developed for the purpose of perspicuously describing particular areas of linguistic theory.

9.1.2. The solution

The solution to the problem of interfacing AI problem-solving techniques and an established linguistic formalism has involved identifying a linguistic formalism that in fact uses the same representation as the required problem-solving methods. Ironically, this linguistic formalism originated not from mathematics or from the theory of computation, but from anthropology. The formalism is Halliday's systemic grammar, which originated from the work of the anthropologist Malinowski and the sociolinguist Firth.

The shared representation, which is the basis of this solution, resulted from the historical accident that at the core of both AI problem solving and systemic grammar is the representation of a space of alternatives. In each case this representation consists of describing the conditions under which an alternative is appropriate, and the effects or consequences of that alternative. Systemic grammar is probably unique in having this fundamental relationship with AI problem solving because the emphasis on paradigmatic description is an invention of Firth and Halliday.

The common representation means that in fact no interface per se is needed at all. An AI problem solver can simply interpret a systemic grammar as linguistic knowledge to be used to solve linguistic problems, in exactly the same way as it can use chemistry to solve chemistry problems, or medical knowledge to solve medical problems. Indeed, it is only because there is no interface per se that the solution is possible--otherwise the powerful computational techniques embodied in the problem solver would lose control to the

interface component during the processing of the grammar.

A significant advantage of SLANG is that it embodies linguistically, and exploits computationally, Halliday's semantic stratum of systemic theory. Computationally, the semantic stratum acts as a body of large-grain-size compiled knowledge that guides the problem solving at the grammatical stratum. Thus the semantic stratum serves linguistically to link the grammar to the social situation, and computationally to increase significantly the speed of the text generation.

The resulting approach to text generation, then, uses the state-of-the-art computational techniques (e.g. forward-chaining, goal-directed backward-chaining, and knowledge compilation), and an explicit grammar--including the semantics--represented in an established linguistic formalism.

9.1.3. Theoretical issues

Chapter 5 examined some of the theoretical issues raised by the SLANG approach. The three areas of interest are: the interface issues, the functional nature of systemic grammar, and the relationship between systemic/functional grammar and the currently dominant generative paradigm.

There are two different interfaces in SLANG worth noting: the interface between the text planner and the text generator, and the interface between the semantics and the grammar. The first of these, as mentioned above, is not an interface *per se*; the only distinction that could be made is between text-planning *knowledge* and text-generation *knowledge* since the same inference engine uses both. Even the distinction between these is not clear-cut because the boundary between them depends on the amount of compilation of the knowledge in the semantic stratum. The second interface--between the semantics and the grammar--is of interest because of what appear to be grammatical structural descriptions in the preselection realization rules at the semantic stratum. These structural descriptions were shown to be part of a legitimate interface between the two

strata, and the modularity between the two is not violated.

Systemic grammar, as a functional grammar, contrasts with the more popular formal approach. However, it was argued that the relationship between the functional and formal approaches to language correspond to the functional and formal approaches to biology--physiology and anatomy respectively. It was then pointed out that the study of process in language, like the study of process in biology, may benefit from a functional approach. Specifically, issues such as the social aspects of language and language evolution can be studied; and in the context of text generation, teleological explanations can be given for the texts and the processes that generated them.

There are two important ways in which SLANG is at odds with the generative paradigm. The first of these concerns the modularity hypothesis made in the generative paradigm: that there is a language faculty that operates relatively independently of the rest of the cognitive mechanism. SLANG does not respect this hypothesis and takes advantage of powerful, general-purpose AI representations and techniques. This bears on the second theoretical issue: the power of the grammar. Since SLANG exploits general problem-solving representations and methods, the resulting model is far too powerful to be of interest from a generative point of view. This is not to be seen as a liability--the search for general theories of representation and problem solving, and the "identification of general principles and mechanisms that underlie all thought processes," form an integral part of the paradigm of computational linguistics (see Winograd, 1983, p. 186).

9.1.4. The formal model

One of the interesting results of the fundamental relationship between AI problem solving and systemic grammar is the formal model presented in Chapter 6. Systemic grammar has always lacked the formal treatment available for grammars of mathematical origin. However, since the common representation allows a systemic grammar to

be interpreted as a set of productions (a common AI representation), an almost traditional formalization in terms of productions can be given.

A formal treatment of the SLANG model can then be given in terms of the formalized grammar and formal algorithms from the AI literature (in particular, problem-reduction). Several lemmas can then be proven concerning both completeness and the relationship between the grammar and the algorithms. This formal model of systemic grammar is particularly significant because it demonstrates that systemic grammar can be rigorously formalized in terms of rules, while allowing the grammar to describe language as a resource.

9.1.5. The implementation

For the purposes of testing and demonstrating SLANG, a prototype system was constructed using a grammar pieced together from several sources. Several shortcuts were taken to keep the project manageable: the phonological/orthographic stratum, the clause-complex rank and the morphological rank were omitted, the word rank is small and *ad hoc*, and only a very small semantic stratum has been implemented. Nevertheless, the system adequately shows the processing of a large system network (the clause systems), interstratal preselection, and inter-rank preselection. The system was tested by generating examples (see Appendix B) from several domains. These include two sets of examples that illustrate generation from explicit semantic networks (listed in Appendix C), supporting the plausibility of the proposed generation method.

The test system, SLANG-I, is written in the production language OPS5. The systemic grammar is stored as a set of OPS5 productions after being translated from a LISP-based system network notation. The grammar can then be loaded into OPS5 and used directly to to do forward- and backward-chaining. The text-generation system itself is very simple, consisting of a small number of productions and LISP operators to build linguistic structures from realization rules and to do low-level maintenance.

9.1.6. Other work

Finally, a comparison was made between SLANG and some other recent text-generation projects. It was observed that all the projects could be classified as either grammar-driven, goal-driven, or an attempt to combine the two. Grammar-driven systems have two main advantages: the grammars can be understood, judged, and modified independently of the computational aspects of the system; and the grammar provides an explicit and useful display of the logical structure of the text-generation process. Goal-driven systems have three main advantages: they exploit the powerful goal-directed AI problem-solving techniques; they allow linguistic and non-linguistic acts to be integrated; and they allow the semantics to be organized independently of the grammar.

Davey's PROTEUS and Mann's Nigel are clearly grammar-driven systems, Appelt's KAMP and McDonald's MUMBLE are clearly goal-driven systems, while Appelt's TELEGRAM and SLANG are attempts to combine the grammar-driven and goal-driven approaches to achieve the advantages of both.

Although TELEGRAM was an attempt to combine the grammar-driven and goal-driven approaches, it fell short of this aim because of the problem of interfacing the AI problem-solving methods with the functional unification formalism. The systemic grammar in SLANG, on the contrary, can be used directly as linguistic problem-solving knowledge by a state-of-the-art AI problem solver. Thus SLANG is able to achieve successfully the advantages of both the grammar-driven and goal-driven approaches.

9.2. Major problems

Though the preceding arguments have hopefully convinced the reader that SLANG represents a promising new approach to text generation, there are still some problems to be overcome before this promise can be fully realized. These problems are not computational--given only the existing computational methodologies, and existing hardware and software tools, SLANG should be able to

provide an effective, practical text-generation facility, given sufficient linguistic resources. The problem is that sufficient linguistic resources do not yet exist.

The relatively small number of linguists working within the systemic framework has severely restricted the availability and coverage of the grammars. Only small subsets of English are covered, even for areas that have received the most attention (e.g. the clause). Large components of the grammatical description have been left untouched (for instance, it seems no substantial work has been done on the systemic morphology of English). Often the level of detail of grammars in the linguistic literature is not suitable for computational treatment.

The grammar developed by Mann et al. (1983) and the grammar pieced together for SLANG-I indicate that at least for the grammatical stratum, the large system networks necessary for practical text generation can be constructed. The doubt really lies with the semantic stratum. But even here, one of the most difficult issues--the organization--has already been resolved. The semantic stratum is a paradigmatic description of register, as prescribed by Halliday (1978).

The obvious application for SLANG is text generation in expert systems. Some linguistic work on particular expert-system domains has been done within the systemic framework (e.g. Mishler, 1984), but unfortunately no sufficiently detailed semantic analyses of expert system domains are currently available. Nevertheless, several considerations in the field of expert systems justify an optimistic outlook on the semantic stratum.

Expert systems--almost by definition--work in very restricted domains (see Brachman et al., 1983, p. 42). In many cases at least, this implies that the linguistic registers involved will also be very restricted. The high degree of specialization that makes the non-linguistic domain knowledge manageable yet practical, may also make the semantic knowledge manageable yet practical.

Another point is that the semantic stratum could potentially be developed to a large extent during the normal knowledge-acquisition-engineering process when building an expert system. Note that linguistic observation is already part of the recommended procedure:

> In addition, the knowledge engineer also listens for justifications of the associations, terms, and strategical methods the expert uses when solving a problem. These are important to record not only for the knowledge engineer's own clarification but also for maintaining adequate system documentation and allowing accurate system explanations. (Buchanan et al., 1983, p.135)

> The knowledge engineer schedules numerous meetings with the expert over a period of a few months to uncover the basic concepts, primitive relations, an definitions needed to talk about the problem and its solutions. (ibid., p. 133)

> Record a detailed protocol of the expert solving at least one prototypical case.
> ... It provides a list of vocabulary terms and hints about strategies. (ibid., p. 161)

In addition to a study of the terms the expert uses in particular registers, the grammatical constructions used in those registers could also be noted.

No doubt the construction of semantic strata will be slow and painful in the beginning. But, as more experience accumulates, and generalizations are passed on, and techniques are developed, building a semantic stratum may become no worse than any other area of knowledge acquisition and engineering.

9.3. Future research

Given the exploratory nature of this work, it is not surprising that there are several interesting continuations and offshoot ideas that can be pursued in future research. Two obvious continuations--the incorporation of SLANG into a full-scale expert system, and the improvement of the currently implemented linguistic capabilities--will be touched on first. Then a technical offshoot concerning highly-compiled semantics will be looked at. Finally, the

possibility and implications of the SLANG ideas for parsing will be explored.

9.3.1. Incorporation of SLANG into an expert system

One obvious continuation of the present work is to incorporate SLANG as the text-generation component of an expert system. This would allow SLANG to be studied in the context of a full-blown AI problem solver, and would provide an opportunity to examine the requirements of the semantic stratum objectively.

9.3.2. Supplementary linguistic treatment

9.3.2.1. Systemic speech generation

An interesting and practical extension to this work would be to implement the phonological stratum and use SLANG to do speech generation. Halliday already has a relatively well-developed treatment of phonology, especially intonation and rhythm, and its relation to the rest of the linguistic system (e.g. see Halliday, 1976c; and Halliday, 1985, Chapter 8). This would also involve augmenting the grammatical stratum. Specifically, another functional analysis would have to be included: the analysis of Given and New (see also Halliday, 1985, pp. 274-281 and Winograd, 1983, p. 284-285) as part of the textual metafunction. Realization rules would have to be added to the grammatical stratum to preselect appropriate phonological features.

9.3.2.2. Supply missing grammatical ranks

Although the grammatical stratum forms the bulk of the implementation, shortcuts have been taken in the current implementation. Both the top and bottom ranks have been omitted--neither the clause-complex nor the morphological rank has been implemented. The former has not been needed because the examples have been so small, and the latter has been avoided by greatly constraining the size of the dictionary and listing all morphological forms explicitly. It

is clear, however, that in a larger system--even a larger prototype--both of these ranks would be absolutely necessary.

9.3.2.3. Linguistic defaults

Another potentially useful idea is the unmarked feature. These appear often in the systemic literature, and are indicated by an asterisk (e.g. Figure 4.6). The idea is that if the entry conditions for a system are satisfied, and none of the other features have been chosen, then the unmarked feature is chosen by default. The usefulness of this particular mechanism is suggested by the interest in default reasoning in AI problem solving (e.g. Stefik et al., 1983, p. 73). A set of productions, one for each default feature, could be introduced which have the entry conditions of the system as conditions and the choice of the features as the effect. Some mechanism would also have to be added to make sure all other avenues of reasoning are explored before any default production fires. It is not clear if the advantages of such default reasoning outweigh the added complexity.

9.3.3. Further compilation of the semantics

The semantic stratum was described in Chapter 4 as "compiled knowledge" that guides the problem solving at the grammatical stratum. It was pointed out (Section 4.2.3) that the knowledge at the semantic stratum could potentially be compiled further by precomputing the inference at the grammatical stratum and attaching the results directly to the semantic features in the internal representation. For the purposes of discussion, this further compilation will be called "hypercompilation."

The process of hypercompilation would involve looking at each preselection statement at the semantic stratum, precomputing the resulting forward- and backward-chaining as far as possible (including following any preselection statements leading to lower ranks and strata), and collecting the realization rules attached to the features involved. The realization rules are then attached directly

to the original semantic feature in place of the preselection rule. All this is done automatically, and is transparent to the grammar writer. This means that very sophisticated computational techniques can be used during the hypercompilation, including keeping detailed tables, carefully looking for redundant realization rules, and so on. If the linguist makes a change to the system networks at any stratum, the hypercompilation is done again, perhaps incrementally. It may be desirable for the computational linguist to designate only certain portions (e.g. high frequency registers) of the semantic stratum for hypercompilation.

The idea of hypercompilation is very vague at this stage, but it may be worth investigating. It appears that it may allow a significant increase in the speed of the text generation, while being linguistically transparent.

9.3.4. Reasoning with knowledge at the grammatical stratum

Another topic related to knowledge compilation is the possibility of "reasoning from first principles" (see Section 2.2.2) with the knowledge at the grammatical stratum. It is possible that the grammatical stratum could be used as base-level knowledge to solve problems when there is no appropriate compiled knowledge available (insufficient semantic knowledge) or to provide automatic linguistic explanations of generated text, since the use of compiled knowledge prevents the automatic generation of teleological explanations.

One example of this kind of reasoning--solving the goal of conflating the Agent and the Theme--has already been described in detail (4.2.2). It is unclear, however, how far this type of reasoning can be taken.

9.3.5. Natural-language understanding

The final point for future research is to investigate the implications of the present work for natural-language understanding. The major problem that has been solved here is how to interface AI problem solving with a linguistic formalism. It was shown how the

higher levels of linguistic and non-linguistic knowledge could be used to process the grammar selectively and efficiently. The main point to be made in this section is that the same problem occurs in natural-language understanding, and that the same solution also may apply.

Parsing, like text generation, is a task that requires substantial guidance from higher-level knowledge. Parsing also preferably uses an established linguistic formalism. Thus the same interface problem that occurs in text generation also occurs in parsing. The solution to the problem in text generation may also work for parsing.

The semantic stratum, during a parse, can preselect features from the grammatical stratum. In this case the preselections represent hypotheses instead of goals, but the same forward- and backward-chaining can be done to determine all the implications of these hypotheses. The suggestion that the semantic stratum, representing knowledge of register, could be useful when parsing should not be surprising. As Halliday says, given a particular register

> ... we can predict quite a lot about the language that will be used, in respect of the meanings and the significant grammatical and lexical features through which they are expressed. If the entries under field, tenor and mode are filled out carefully and thoughtfully, it is surprising how many of the features of the language turn out to be relatable to the context of situation. This is not to claim that we know what the participants are going to say; it merely shows that we can make sensible and informed guesses about certain aspects of what they might say, with a reasonable probability of being right. There is always, in language, the freedom to act untypically--but that in itself serves to confirm the reality of the concept of what is typical. (Halliday, 1978, p. 226)

If the register could only ascertained by linguistic means, then using knowledge of register to help understand language would be begging the question. This is not the case however. Register is largely determined from outside language. Field and tenor in particular, are determined, to a large extent, before any linguistic

interaction occurs at all. The physical setting, the social status of the participants, and even the emotional issues at the moment (see Section 3.5), can often be ascertained easily without linguistic interaction.

> The linguistic system ... is organized in such a way that the social context is predictive of the text. This is what makes it possible for a member to make the necessary predictions about the meanings that are being exchanged in any situation which he encounters. If we drop in on a gathering, we are able to tune in very quickly, because we size up the field, tenor and mode of the situation and at once form an idea of what is likely to be being meant. In this way we know what semantic configurations--what register--will probably be required if we are to take part. If we did not do this, there would be no communication, since only a part of the meanings we have to understand are explicitly realized in the wordings. The rest are unrealized; they are left out--or rather (a more satisfactory metaphor) they are out of focus. We succeed in the exchange of meanings because we have access to the semiotic structure of the situation from other sources. (ibid., p. 189)

Knowledge of register is not used in any substantial or systematic way by current natural-language understanding systems. Some systems use knowledge of the world to resolve the ambiguity of input texts (e.g. Winograd, 1972), and other systems use knowledge of discourse structure and intentions (Grosz and Sidner, 1985). But these are not using knowledge of what linguistic devices are specific to what social situations. Winograd's program does not take advantage of the fact that there are very specific English constructs used to describe the relations between objects like blocks and the operations that are performed on sets of these objects (knowledge of field).*

Neither does Winograd's program take advantage of the relationship between the "robot" and the interlocutor (knowledge of tenor). The program should, to use a simple example, be expecting imperatives. Some systems (e.g. Sullivan and Cohen, 1985) use knowledge

---

* No doubt Winograd took advantage of this when constructing his grammar and dictionary, but the program does not.

of the relationship between the speaker and hearer to make inferences about the speaker's intentions. However, these systems do not use knowledge of the specific linguistic constructs likely to appear as a result of this relationship.

Similarly, using knowledge of discourse patterns is not the same as using knowledge of which patterns of discourse are used in which types of social situations (knowledge of mode--for instance, knowledge of the ellipsis that is used heavily in question and answer dialogues). Knowledge of the most likely types of reference in particular situations would also be useful.

Unlike text generation, the parsing interface works both ways--the results of the parse must be passed back up through the semantics. It may be possible to do the sort of reconstructive reasoning that MYCIN, for example, does (Hasling et al., 1984). Once a set of grammatical features has been determined, whose realization rules specify a syntactic structure corresponding to the text, the semantic stratum is examined to find semantic features compatible with the register which preselect the grammatical seed features.

Only a few natural-language understanding issues have been mentioned here and none have been examined in detail. It appears, however, that a more thorough investigation of the implications of SLANG for natural-language understanding would be worthwhile.

9.4. Conclusion

Text generation is a subfield of natural-language work that has received relatively little attention. However, as expert systems move into areas such as medicine and law, where effective natural-language communication is important, text-generation research acquires a new significance.

This thesis has presented a novel approach to text generation--the Systemic Linguistic Approach to Natural-language Generation (SLANG). By interpreting a systemic grammar as AI problem-solving knowledge, SLANG is able to breach one of the major obstacles in

text generation, viz. how to interface computational AI problem-solving techniques with an established linguistic formalism. Thus SLANG, unlike any other approach to date, allows that text generation to be performed by a powerful, goal-directed AI problem solver, while the process still follows an explicit grammar written in an established linguistic formalism.

Although SLANG has been formalized and implemented, its status at best is that it has successfully undergone a preliminary investigation. Much more work will have to be done to discover the real limitations, or for that matter the real benefits, of this approach.

Appendix A: Introduction to OPS5

OPS5 (Forgy, 1981; Brownston et al., 1985) is the production language used to implement SLANG-I. The purpose of this appendix is to provide the reader with enough OPS5 background to understand the OPS5 terms and code that appear in the implementation chapter.

The two main components of an OPS5 production system are the production memory, which stores the productions themselves, and the working memory, which is a repository of information accessed and modified by the productions. Productions have a left-hand-side (LHS) and a right-hand-side (RHS). The LHS is a list of patterns which is "matched" if all the patterns are matched by working memory elements. The RHS is a list of procedure calls.

When OPS5 is running, it takes all the productions whose LHS is matched by the current working memory, selects the production which is matched by the most recent working memory elements, and executes the procedure calls in the RHS of that production. A new set of productions whose LHS is matched is then calculated and the process repeats. This process begins with a working memory initiallized by the user, and ends when the set of productions whose LHS is matched is empty.

Pattern matching

The patterns in the LHS of productions and in working memory are of two varieties: attribute-value pairs and vectors. An attribute-value pair pattern is of the form (identifier ^a1 v1 ^a2 v2 ... ^an vn) where the order of the individual pairs ^ai vi is not significant. For instance, there may be an element in working memory (house ^colour white ^floors 2 ^rooms 10). LHS patterns matching this element must have the same identifier and some (possibly empty) subset of the attribute-value pairs in any order. For instance, (house ^colour white) and (house ^rooms 10 ^floors 2) would match but (house ^colour blue) would not.

The values can be made more general through the use of the symbols for "not equal" "<>", "greater than" ">", and so on. For instance (house ^colour <> pink), (house ^rooms > 4 ^floors < 4) would match the description of the house above, but (house ^rooms < 8) would not. Disjunction can be specified with double angle brackets "<<" and ">>"--e.g. (house ^colour << green white red >> ^floors << 1 2 >>).

OPS5 rules look like

```
(p production-name
   LHS
-->
   RHS)
```

So an actual production might be

```
(p eg1
  (house ^colour << red white >> ^rooms > 6)
-->
  ...
```

The LHS of this production would be satisfied if there is an element in working memory describing a red or white house with more than 6 rooms.

It is usually necessary to have variables in productions to link the different patterns in the LHS and to mediate between the LHS and the RHS. Variables in OPS5 are symbols whose first and last characters are open and closed angle brackets respectively (e.g. <x>, <house1>, <new-house> etc.). The LHS of the production

```
(p eg2
  (house ^colour <house-colour>
         ^rooms <house-rooms>)
  (customer ^age > 25
            ^eyes <house-colour>
            ^children < <house-rooms>)
-->
  ...
```

will be matched if there are elements in working memory describing a house, and a customer who is older than 25, whose eyes are the same colour as the house, and who has fewer children than the number of

rooms of the house. Several constraints can be put on a value using curly brackets "{}". For instance

```
(house ^colour {<> blue <> red <house-colour>}
       ^rooms {< 12 <house-rooms> > 5})
```

will match a working memory element describing a house which is not blue or red and has between 6 and 11 rooms--setting the variable <house-colour> to whatever the actual colour is, and setting the variable <house-rooms> to whatever the number of rooms is.

The other type of pattern used in OPS5 is the vector. In this case there are no attributes; the vector is simply a list of values where order is important. A LHS vector pattern with n symbols is matched by a working memory vector whose first n elements match those of the pattern. The vector (roses are << red black >> and <flowers> are <colour2>) is matched by (roses are red and violets are blue), (roses are black), and (roses).

The RHS

The RHS of an OPS5 production is a list of procedure calls of the form (procedure arg1 arg2 ...). The procedure must be a built-in OPS5 procedure. The procedures used in SLANG are "make", "modify", "remove", and "call".

The procedure "make" is used to add an element (attribute-value or vector) to working memory. For instance, (make house ^colour white ^rooms 10 ^floors 2) and (make roses are red) add the working memory elements (house ^colour white ^rooms 10 ^floors 2) and (roses are red) respectively.

The procedure "modify" is used to change an existing element in working memory. The part to be modified is specified by an attribute in the case of attribute-value pairs, or an index number in the case of vectors. The working memory element to be modified is identified by the number of the LHS pattern it matches (1 to modify a match of the first pattern, 2 to modify a match of the second pattern and so on) or a label on the pattern. For instance if the

patterns of

```
(p eg3
  (house ^id <h>)
  (paint <h> <new-colour>)
-->
  (modify 1 ^colour <new-colour>)
  (modify 2 ^1 painted))
```

are matched by (house ^id house3 ^colour white) and (paint house3 black), and if OPS5 fires this production, then these working memory elements will be changed to (house ^id house3 ^colour black) and (painted house3 black). The same production could also be written using labels on the patterns:

```
(p eg3
  {(house ^id <h>)              <house>}
  {(paint <h> <new-colour>)     <paint>}
-->
  (modify <house> ^colour <new-colour>)
  (modify <paint> ^1 painted))
```

<house> and <paint> above are "element variables" that represent the entire working memory element that matches the pattern.

The procedure "remove" simply deletes a working memory element identified in either of the ways described above for "modify"--e.g. (remove 2) or (remove <paint>).

The procedure "call" is used to call a procedure the user has defined in another language (e.g. LISP). The parameter-passing conventions are awkward to explain, but the parameters themselves are either identifiers, variables, or the results of function calls (see below). The procedures do not pass back values, but access working memory directly through some special routines provided by OPS5.

The procedure "write" simply writes its arguments to the terminal. It too takes identifiers, variables and function calls as arguments. The function often called from "write" is "crlf," which prints a carriage return at the terminal.

Besides "crlf", the only other functions called in SLANG-I are "substr" and "genatom." The function "substr" returns a substring of

an element matched in the LHS. The only call on this function is (call PRESELECT (substr <preselect> 1 inf)), which passes the entire working memory element matching the element variable <preselect> to the LISP operator PRESELECT. The function "genatom" takes no arguments and returns a unique identifier--this is essentially the same as "gensym" in LISP.

Appendix B: Sample texts

The following are some examples of the text produced by SLANG-I using the grammar in Appendix C. Recall that SLANG-I was not provided with an orthographic stratum, so there are no "an"s, punctuation or capitalization--except as provided by the systems at the word rank (the dictionary). To avoid confusion, line spacing has been added in lieu of punctuation where necessary.

In all cases SLANG-I generates the text one clause at a time. Although some of the samples are of paragraph length, it should not be inferred that SLANG-I has done any text planning; all the examples are collections of clauses which, as far as SLANG-I is concerned, were generated independently.

Note that only the final set of samples uses the semantic stratum listed at the end of Appendix C.

1. Explanation for a hypothetical expert system

The following example was generated to demonstrate the grammar, and to illustrate the utility of flexible natural-language generation in expert systems.

Suppose there is a hypothetical medical expert system interviewing the mother of a patient named Mary (following an example in Hasling et al., 1984). The mother has reported that Mary has been suffering from stiff neck muscles and headaches. At this point the hypothetical dialogue continues:

Does Mary have a fever?
*WHY

Mary's mother wants to know why she is being asked this question. The following text was generated by preselecting the grammatical features by hand. The construction of a good semantic stratum in this domain would be a major project in itself.

well Mary has been having headaches
on this basis perhaps she has a infection
this possibility would be supported by a fever
so we ask
does she have one

The preselections for each of the clauses is as follows:

CLAUSE 1.

```
($C1<Carrier<Head : !mary)
($C1<Carrier : non-possessive-nom)
($C1<Carrier : noun)
($C1<Carrier : non-determined)
($C1<Carrier : non-quantified)
($C1<Carrier : singular)

($C1<Attribute<Head : !headache)
($C1<Attribute : plural)
($C1<Attribute : noun)
($C1<Attribute : non-determined)
($C1<Attribute : non-quantified)
($C1<Attribute : non-possessive-nom)

($C1 : possessive-attribute)
($C1 : ascriptive)
($C1 : range-operative)
($C1 : residual)
($C1 : present)
($C1 : past-in)
($C1 : present-in)
($C1 : unmarked-declarative-theme)
($C1 : singular-subject)
($C1 : nominal-subject)
($C1 : non-attitudinal)
($C1 : textual-theme)
($C1 : responsive/explanative)
($C1 : unmarked-positive)
($C1 : non-place)
($C1 : non-time)
```

CLAUSE 2.

```
($C2<Carrier<Head : !feminine)
($C2<Carrier : non-possessive-nom)
($C2<Carrier : personal)
($C2<Carrier : singular)

($C2<Attribute<Head : !infection)
($C2<Attribute : singular)
($C2<Attribute : noun)
($C2<Attribute : determined)
($C2<Attribute : non-add)
($C2<Attribute : non-quantified)
($C2<Attribute : non-possessive-nom)
($C2<Attitude : !obvious)
($C2<Attitude : !low)

($C2 : possessive-attribute)
($C2 : ascriptive)
($C2 : pronominal-subject)
($C2 : range-operative)
($C2 : residual)
($C2 : present)
($C2 : non-past-in)
($C2 : non-present-in)
($C2 : unmarked-declarative-theme)
($C2 : singular-subject)
($C2 : pronominal-subject)
($C2 : non-place)
($C2 : non-time)
($C2 : interpersonal-theme)
($C2 : textual-theme)
($C2 : prop-reason)
($C2 : unmarked-positive)
```

CLAUSE 3.

```
($C3<Medium : noun)
($C3<Medium : singular)
($C3<Medium : determined)
($C3<Medium : non-quantified)
($C3<Medium : non-possessive-nom)
($C3<Medium : non-add)
($C3<Medium<Head : !fever)

($C3<Process : !support)
```

```
($C3<Range : noun)
($C3<Range : singular)
($C3<Range : determined)
($C3<Range : non-quantified)
($C3<Range : near)
($C3<Range<Head : !possibility)
($C3<Range : non-possessive-nom)

($C3<Modal : !would)

($C3 : non-past-in)
($C3 : non-present-in)
($C3 : unmarked-positive)
($C3 : modal)
($C3 : unmarked-declarative-theme)
($C3 : non-attitudinal)
($C3 : non-textual-theme)
($C3 : mediated)
($C3 : residual)
($C3 : singular-subject)
($C3 : nominal-subject)
($C3 : non-place)
($C3 : non-time)
```

CLAUSE 4.

```
($C4 : speaker-plus-subject)
($C4 : unmarked-positive)
($C4 : unmarked-declarative-theme)
($C4 : non-attitudinal)
($C4 : textual-theme)
($C4 : residual)
($C4 : interrogating)
($C4 : gen-simple)
($C4 : prop-causal)
($C4 : present)
($C4 : non-past-in)
($C4 : non-present-in)
($C4 : non-place)
($C4 : non-time)

($C4<Process : !ask)
```

CLAUSE 5.

```
($C4<Beta : present)
($C4<Beta : non-past-in)
($C4<Beta : residual)
($C4<Beta : non-present-in)
($C4<Beta : singular-subject)
($C4<Beta : range-operative)
($C4<Beta : unmarked-yes/no-theme)
($C4<Beta : non-textual-theme)
($C4<Beta : ascriptive)
($C4<Beta : possessive-attribute)
($C4<Beta : unmarked-positive)
($C4<Beta : pronominal-subject)
($C4<Beta : non-place)
($C4<Beta : non-time)

($C4<Beta<Carrier : non-possesive-nom)
($C4<Beta<Carrier : personal)
($C4<Beta<Carrier : singular)
($C4<Beta<Carrier<Head : !feminine)

($C4<Beta<Attribute : non-possessive-nom)
($C4<Beta<Attribute : singular)
($C4<Beta<Attribute : non-determined)
($C4<Beta<Attribute : non-quantified)
($C4<Beta<Attribute : substitute)
```

2. Sample explanation of a plan

The following text was generated as part of a project to automatically generate explanations of plans. The program that did the plan analysis and text planning (Sothcott, 1985) also does the semantic reasoning, so again only the grammatical stratum was used by SLANG-I. It is assumed that the planner is explaining the plan to one of the plan participants who is responsible for the sanding, painting and varnishing.

first you do the painting

at the same time the basement floor is poured

at the same time the plasterer fastens the plaster board

if the basement floor has been poured
and the plaster board has been fastened
then the finished flooring can be laid

after that the carpentry can be finished

if the carpentry has been finished
and you've done the painting
then you can sand the floors

after that you can varnish the floors

The text planner EXPLAN (Sothcott, 1985) begins by running a scheduler that examines the output of a planner and produces a schedule of the planned actions to be described. The text planner then decides what sort of description should be given for each of these actions. The result is a blueprint for the entire text in terms of a high-level description for each clause. The text planner then fills the functional role slots for each clause using domain-specific knowledge and input from the user. Based on relationships between these functional roles (e.g. if the Actor and the Subject match), and domain specific knowledge about processes and entities (e.g. knowledge that "carpentry" is a mass entity), it preselects appropriate grammatical features. The following are the preselection lists for the above example as generated by EXPLAN (with very slight modifications to allow them to run on a later version of the grammar).

```
(comment
Addressee: painter
Discourse type: actor_focussed

)

(comment

Node number: 9
Sentence number: 1
Clause number: 1
Clause type: core
Topical theme: painter
Subject: painter
Voice: operative
Case frame:
        [Goal painting]
        [Process do]
        [Actor painter]


)

($C<Goal<Head : !painting)
($C<Goal : non-possessive-nom)
($C<Goal : non-quantified)
($C<Goal : non-selective)
($C<Goal : determined)
($C<Goal : noun)
($C<Goal : mass)
($C<Process : !-do-)
($C : unmarked-imperative-theme)
($C : imperative-subject-explicit)
($C : dispositive)
($C : non-benefactive)
($C : residual)
($C : operative)
($C : unmarked-positive)
($C : non-attitudinal)
($C : present)
($C : non-present-in)
($C : non-past-in)
($C : thesis-initial)
($C : textual-theme)
($C : non-place)
;----------------------------------------
```

```
(comment

Node number: 5
Sentence number: 2
Clause number: 2
Clause type: core
Topical theme: basement_floor
Subject: basement_floor
Voice: non-agentive  (passive)
Case frame:
        [Goal basement_floor]
        [Process pour]
        [Actor unknown]


)

($C<Goal<Head : !basement-floor)
($C<Goal : non-possessive-nom)
($C<Goal : non-quantified)
($C<Goal : non-selective)
($C<Goal : determined)
($C<Goal : noun)
($C<Goal : singular)
($C<Process : !pour)
($C : unmarked-declarative-theme)
($C : singular-subject)
($C : nominal-subject)
($C : creative)
($C : non-benefactive)
($C : non-residual)
($C : non-agentive)
($C : unmarked-positive)
($C : non-attitudinal)
($C : non-present-in)
($C : non-past-in)
($C : present)
($C : thesis-simultaneous)
($C : textual-theme)
($C : non-place)
;----------------------------------------
```

```
(comment

Node number: 4
Sentence number: 3
Clause number: 3
Clause type: core
Topical theme: plasterer
Subject: plasterer
Voice: operative
Case frame:
        [Goal plaster_board]
        [Process fasten]
        [Actor plasterer]


)

($C<Goal<Head : !plaster-board)
($C<Goal : non-possessive-nom)
($C<Goal : non-quantified)
($C<Goal : non-selective)
($C<Goal : determined)
($C<Goal : noun)
($C<Goal : mass)
($C<Actor<Head : !plasterer)
($C<Actor : non-possessive-nom)
($C<Actor : non-quantified)
($C<Actor : non-selective)
($C<Actor : determined)
($C<Actor : noun)
($C<Actor : singular)
($C<Process : !fasten)
($C : unmarked-declarative-theme)
($C : singular-subject)
($C : nominal-subject)
($C : dispositive)
($C : non-benefactive)
($C : residual)
($C : operative)
($C : unmarked-positive)
($C : non-attitudinal)
($C : non-present-in)
($C : non-past-in)
($C : present)
($C : thesis-simultaneous)
($C : textual-theme)
($C : non-place)
;----------------------------------------
```

```
(comment

Node number: 5
Sentence number: 4
Clause number: 4
Clause type: subsidiary
Topical theme: basement_floor
Subject: basement_floor
Voice: non-agentive  (passive)
Case frame:
        [Goal basement_floor]
        [Process pour]
        [Actor unknown]


)

($C<Goal<Head : !basement-floor)
($C<Goal : non-possessive-nom)
($C<Goal : non-quantified)
($C<Goal : non-selective)
($C<Goal : determined)
($C<Goal : noun)
($C<Goal : singular)
($C<Process : !pour)
($C : unmarked-declarative-theme)
($C : singular-subject)
($C : nominal-subject)
($C : creative)
($C : non-benefactive)
($C : non-residual)
($C : non-agentive)
($C : unmarked-positive)
($C : non-attitudinal)
($C : thesis-conditional)
($C : cond-antecedent)
($C : present)
($C : past-in)
($C : non-present-in)
($C : textual-theme)
($C : non-place)
($C : non-time)
;----------------------------------------
```

```
(comment

Node number: 4
Sentence number: 4
Clause number: 5
Clause type: subsidiary
Topical theme: plaster_board
Subject: plaster_board
Voice: non-agentive  (passive)
Case frame:
        [Goal plaster_board]
        [Process fasten]
        [Actor plasterer]


)

($C<Goal<Head : !plaster-board)
($C<Goal : non-possessive-nom)
($C<Goal : non-quantified)
($C<Goal : non-selective)
($C<Goal : determined)
($C<Goal : noun)
($C<Goal : mass)
($C<Process : !fasten)
($C : unmarked-declarative-theme)
($C : mass-subject)
($C : nominal-subject)
($C : dispositive)
($C : non-benefactive)
($C : non-residual)
($C : non-agentive)
($C : unmarked-positive)
($C : non-attitudinal)
($C : simp-add)
($C : present)
($C : past-in)
($C : non-present-in)
($C : textual-theme)
($C : non-place)
($C : non-time)
;---------------------------------------
```

```
(comment

Node number: 6
Sentence number: 4
Clause number: 6
Clause type: core
Topical theme: finished_flooring
Subject: finished_flooring
Voice: non-agentive  (passive)
Case frame:
        [Goal finished_flooring]
        [Process lay]
        [Actor unknown]


)

($C<Goal<Head : !finished-flooring)
($C<Goal : non-possessive-nom)
($C<Goal : non-quantified)
($C<Goal : non-selective)
($C<Goal : determined)
($C<Goal : noun)
($C<Goal : mass)
($C<Process : !lay)
($C : unmarked-declarative-theme)
($C : mass-subject)
($C : nominal-subject)
($C : dispositive)
($C : non-benefactive)
($C : non-residual)
($C : non-agentive)
($C : unmarked-positive)
($C : non-attitudinal)
($C : thesis-conditional)
($C : cond-simple)
($C : textual-theme)
($C<Modal : !can)
($C : modal)
($C : non-past-in)
($C : non-present-in)
($C : non-place)
($C : non-time)
;----------------------------------------
```

```
(comment

Node number: 7
Sentence number: 5
Clause number: 7
Clause type: core
Topical theme: carpentry
Subject: carpentry
Voice: non-agentive  (passive)
Case frame:
        [Goal carpentry]
        [Process finish]
        [Actor unknown]


)

($C<Goal<Head : !carpentry)
($C<Goal : non-possessive-nom)
($C<Goal : non-quantified)
($C<Goal : non-selective)
($C<Goal : determined)
($C<Goal : noun)
($C<Goal : mass)
($C<Process : !finish)
($C : unmarked-declarative-theme)
($C : mass-subject)
($C : nominal-subject)
($C : dispositive)
($C : non-benefactive)
($C : non-residual)
($C : non-agentive)
($C : unmarked-positive)
($C : non-attitudinal)
($C : thesis-succeeding)
($C : textual-theme)
($C<Modal : !can)
($C : modal)
($C : non-past-in)
($C : non-present-in)
($C : non-place)
;----------------------------------------
```

```
(comment

Node number: 7
Sentence number: 6
Clause number: 8
Clause type: subsidiary
Topical theme: carpentry
Subject: carpentry
Voice: non-agentive  (passive)
Case frame:
        [Goal carpentry]
        [Process finish]
        [Actor unknown]


)

($C<Goal<Head : !carpentry)
($C<Goal : non-possessive-nom)
($C<Goal : non-quantified)
($C<Goal : non-selective)
($C<Goal : determined)
($C<Goal : noun)
($C<Goal : mass)
($C<Process : !finish)
($C : unmarked-declarative-theme)
($C : mass-subject)
($C : nominal-subject)
($C : dispositive)
($C : non-benefactive)
($C : non-residual)
($C : non-agentive)
($C : unmarked-positive)
($C : non-attitudinal)
($C : thesis-conditional)
($C : cond-antecedent)
($C : present)
($C : past-in)
($C : non-present-in)
($C : textual-theme)
($C : non-place)
($C : non-time)
;---------------------------------------
```

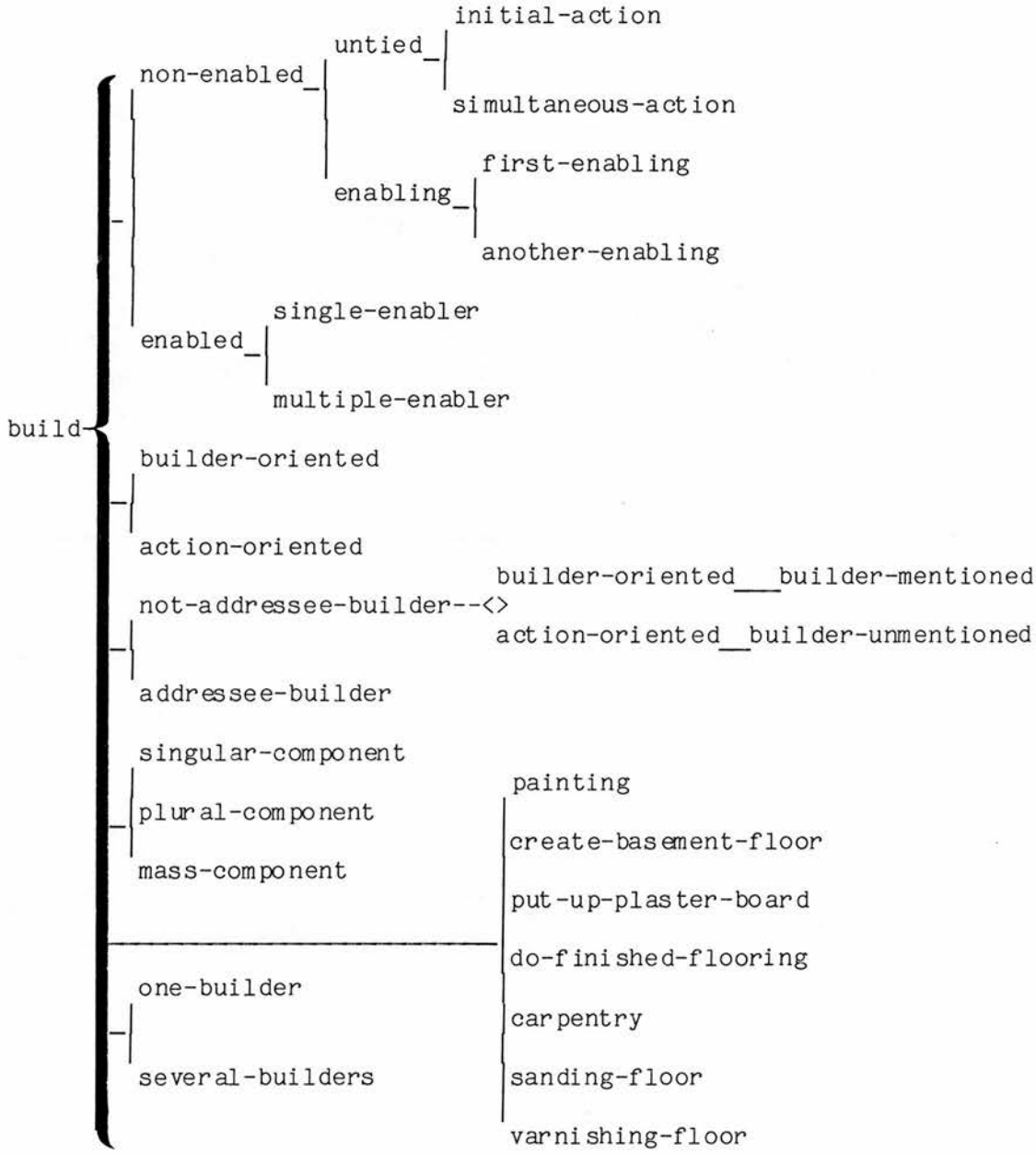
```
(comment

Node number: 9
Sentence number: 6
Clause number: 9
Clause type: subsidiary
Topical theme: painter
Subject: painter
Voice: operative
Case frame:
        [Goal painting]
        [Process do]
        [Actor painter]


)

($C<Goal<Head : !painting)
($C<Goal : non-possessive-nom)
($C<Goal : non-quantified)
($C<Goal : non-selective)
($C<Goal : determined)
($C<Goal : noun)
($C<Goal : mass)
($C<Process : !-do-)
($C : unmarked-declarative-theme)
($C : addressee-subject)
($C : dispositive)
($C : non-benefactive)
($C : residual)
($C : operative)
($C : unmarked-positive)
($C : non-attitudinal)
($C : simp-add)
($C : present)
($C : past-in)
($C : non-present-in)
($C : textual-theme)
($C : non-place)
($C : non-time)
;----------------------------------------
```

```
(comment

Node number: 8
Sentence number: 6
Clause number: 10
Clause type: core
Topical theme: painter
Subject: painter
Voice: operative
Case frame:
        [Goal floors]
        [Process sand]
        [Actor painter]


)

($C<Goal<Head : !floor)
($C<Goal : non-possessive-nom)
($C<Goal : non-quantified)
($C<Goal : non-selective)
($C<Goal : determined)
($C<Goal : noun)
($C<Goal : plural)
($C<Process : !sand)
($C : unmarked-declarative-theme)
($C : addressee-subject)
($C : dispositive)
($C : non-benefactive)
($C : residual)
($C : operative)
($C : unmarked-positive)
($C : non-attitudinal)
($C : thesis-conditional)
($C : cond-simple)
($C : textual-theme)
($C<Modal : !can)
($C : modal)
($C : non-past-in)
($C : non-present-in)
($C : non-place)
($C : non-time)
;---------------------------------------
```

```
(comment

Node number: 2
Sentence number: 7
Clause number: 11
Clause type: core
Topical theme: painter
Subject: painter
Voice: operative
Case frame:
        [Goal floors]
        [Process varnish]
        [Actor painter]


)

($C<Goal<Head : !floor)
($C<Goal : non-possessive-nom)
($C<Goal : non-quantified)
($C<Goal : non-selective)
($C<Goal : determined)
($C<Goal : noun)
($C<Goal : plural)
($C<Process : !varnish)
($C : unmarked-declarative-theme)
($C : addressee-subject)
($C : dispositive)
($C : non-benefactive)
($C : residual)
($C : operative)
($C : unmarked-positive)
($C : non-attitudinal)
($C : thesis-succeeding)
($C : textual-theme)
($C<Modal : !can)
($C : modal)
($C : non-past-in)
($C : non-present-in)
($C : non-place)
;----------------------------------------
```

The interface and compiled knowledge embodied in EXPLAN can be extracted to form a semantic system network for this domain. The blueprint for the text would be constructed exactly as before, but now the text planning of the individual clauses would end by simply setting features of the semantic stratum as goals.

The semantic system network contains the following systems (due to space restrictions, only two gates--builder-mentioned and builder-unmentioned--have been included in the diagram. The complete network including gates and realization rules can be found in Appendix C).



When text planning for a particular action on the schedule, the text planner may decide to describe the other actions that enabled the action in question. When constructing the blueprint for a series of descriptions of enabling actions followed by a description of the enabled action, the planner simply sets goals such as $first-enabling (for the first of the enabling actions) and $multiple-enablers (for the enabled action). If the addressee of the text is the person that is performing the action, then the goal

$addressee-builder is set, otherwise $not-addressee-builder is set. The text planner need not assign grammatical functional roles--that will be done by the grammar. Using the same type of reasoning already done by EXPLAN, the text planner can set the appropriate goals for the number of the component being operated upon, and the operation itself. The EXPLAN choice between Actor-focused and Goal-focused is simply replaced by the choice between $builder-oriented and $action-oriented.

The input SLANG now requires to generate the example is as follows (in each case the semantic features are associated with the top-level hub--by convention hub 0):

first you do the painting

```
(make goal $initial-action 0)
(make goal $action-oriented 0)
(make goal $addressee-builder 0)
(make goal $mass-component 0)
(make goal $one-builder 0)
(make goal $painting 0)
```

at the same time the basement floor is poured

```
(make goal $simultaneous-action 0)
(make goal $action-oriented 0)
(make goal $not-addressee-builder 0)
(make goal $singular-component 0)
(make goal $one-builder 0)
(make goal $create-basement-floor 0)
```

at the same time the plasterer fastens the plaster board

```
(make goal $simultaneous-action 0)
(make goal $builder-oriented 0)
(make goal $not-addressee-builder 0)
(make goal $mass-component 0)
(make goal $one-builder 0)
(make goal $put-up-plaster-board 0)
```

if the basement floor has been poured

```
(make goal $first-enabling 0)
(make goal $action-oriented 0)
(make goal $not-addressee-builder 0)
(make goal $singular-component 0)
(make goal $one-builder 0)
(make goal $create-basement-floor 0)
```

and the plaster board has been fastened

```
(make goal $another-enabling 0)
(make goal $action-oriented 0)
(make goal $not-addressee-builder 0)
(make goal $mass-component 0)
(make goal $one-builder 0)
(make goal $put-up-plaster-board 0)
```

then the finished flooring can be laid

```
(make goal $multiple-enabler 0)
(make goal $action-oriented 0)
(make goal $not-addressee-builder 0)
(make goal $mass-component 0)
(make goal $one-builder 0)
(make goal $do-finished-flooring 0)
```

after that the carpentry can be finished

```
(make goal $single-enabler 0)
(make goal $action-oriented 0)
(make goal $not-addressee-builder 0)
(make goal $mass-component 0)
(make goal $one-builder 0)
(make goal $carpentry 0)
```

if the carpentry has been finished

```
(make goal $first-enabling 0)
(make goal $action-oriented 0)
(make goal $not-addressee-builder 0)
(make goal $mass-component 0)
(make goal $one-builder 0)
(make goal $carpentry 0)
```

and you've done the painting

```
(make goal $another-enabling 0)
(make goal $action-oriented 0)
(make goal $addressee-builder 0)
(make goal $mass-component 0)
(make goal $one-builder 0)
(make goal $painting 0)
```

then you can sand the floors

```
(make goal $multiple-enabler 0)
(make goal $action-oriented 0)
(make goal $addressee-builder 0)
(make goal $plural-component 0)
(make goal $one-builder 0)
(make goal $sanding-floor 0)
```

after that you can varnish the floors

```
(make goal $single-enabler 0)
(make goal $action-oriented 0)
(make goal $addressee-builder 0)
(make goal $plural-component 0)
(make goal $one-builder 0)
(make goal $varnishing-floor 0)
```

There are some important advantages to using a semantic system network as an interface between the text planner and the grammar. The linguistic notation developed expressly for this purpose makes it easier to perceive the grammatically relevant decisions and how they are realized. The modularity of the language production is improved since the text planner is not required to know about grammatical features and functional roles.

3. Examples from the semantic stratum

The following samples were generated using the semantic stratum (based on Halliday, 1978, pp. 82-84), which is listed at the end of Appendix C. The register involves a mother attempting to control the behaviour of her child with a threat.

Each example consists of an input to SLANG-I (a set of semantic seed features to be set as goals), preceded by the resulting text. In each case the features are associated with the top-level hub--by convention hub 0.

```
you'll be going upstairs

(make goal $rejection 0)
(make goal $at-home 0)
(make goal $child-centred-decision 0)
(make goal $deferred 0)


you're going upstairs

(make goal $rejection 0)
(make goal $at-home 0)
(make goal $child-centred-decision 0)
(make goal $pending 0)
(make goal $unmarked-time 0)


I'm taking you upstairs now

(make goal $rejection 0)
(make goal $at-home 0)
(make goal $mother-centred-decision 0)
(make goal $pending 0)
(make goal $immediate 0)


I'll smack you

(make goal $unconditional 0)
(make goal $chastisement 0)
(make goal $smack 0)
(make goal $adult-centred-punishment 0)
```

you mustn't do that because I'll smack you

```
(make goal $threatening-reason 0)
(make goal $non-repetitive 0)
(make goal $chastisement 0)
(make goal $smack 0)
(make goal $explanatory-cond 0)
(make goal $adult-centred-punishment 0)
```

if you do that I'll smack you

```
(make goal $logical-cond 0)
(make goal $non-repetitive 0)
(make goal $chastisement 0)
(make goal $smack 0)
(make goal $adult-centred-punishment 0)
```

don't do that
next time I'll smack you

```
(make goal $straight-threat 0)
(make goal $explicit-repetition 0)
(make goal $chastisement 0)
(make goal $smack 0)
(make goal $exclamatory-cond 0)
(make goal $adult-centred-punishment 0)
```

go upstairs now

```
(make goal $unmarked-command 0)
(make goal $immediate 0)
(make goal $at-home 0)
```

you'll be smacked

```
(make goal $unconditional 0)
(make goal $chastisement 0)
(make goal $smack 0)
(make goal $child-centred-punishment 0)
```

you mustn't do that or you'll be smacked

```
(make goal $threatening-alternative 0)
(make goal $non-repetitive 0)
(make goal $chastisement 0)
(make goal $smack 0)
(make goal $explanatory-cond 0)
(make goal $child-centred-punishment 0)
```

don't do that or you'll be smacked by Daddy

(make goal $threatening-alternative 0)
(make goal $non-repetitive 0)
(make goal $daddy 0)
(make goal $smack 0)
(make goal $exclamatory-cond 0)
(make goal $child-centred-punishment 0)

you mustn't do that
next time I'll smack you

(make goal $straight-threat 0)
(make goal $explicit-repetition 0)
(make goal $chastisement 0)
(make goal $smack 0)
(make goal $explanatory-cond 0)
(make goal $adult-centred-punishment 0)

I am not giving you a sweet

(make goal $deprivation 0)
(make goal $at-home 0)
(make goal $mother-centred-decision 0)
(make goal $pending 0)
(make goal $unmarked-time 0)

you are not being given a sweet

(make goal $deprivation 0)
(make goal $at-home 0)
(make goal $child-centred-decision 0)
(make goal $pending 0)
(make goal $unmarked-time 0)

you will not be being given a sweet

(make goal $deprivation 0)
(make goal $at-home 0)
(make goal $child-centred-decision 0)
(make goal $deferred 0)

I will not be giving you a sweet

(make goal $deprivation 0)
(make goal $at-home 0)
(make goal $mother-centred-decision 0)
(make goal $deferred 0)

Appendix C: The grammar

This appendix contains the grammar used in the implementation, and used to generate the sample texts in Appendix B. The networks were collected from a variety of sources, and patched together when necessary. No attempt has been made to provide a polished grammar--there are several dubious fixes and no doubt many outright errors and omissions. This grammar was intended only to test the ideas described in this thesis.

The notation used here is a LISP-based system network notation described in Section 7.2.1.

1. The clause network

The clause systems were based on (Mann/Halliday), an early version of the clause systems for Nigel (Mann et al., 1983).

```
(nil CLAUSE)

((CLAUSE -[) clause
     (Process / Lexverb) (# ^ Theme))

((clause -{=) MOOD)

((MOOD -[) finite
     (Mood (Finite)) (Mood ^ Residue))

((finite -{=) MOOD-TYPE)

((MOOD-TYPE -[) indicative
     (Mood (Subject)) (Residue ^ #))

((indicative -{=) INDICATIVE-TYPE)

((INDICATIVE-TYPE -[) declarative
     (Subject ^ Finite) (% ^ Subject) (Finite ^ %))

((INDICATIVE-TYPE -[) interrogative)

((interrogative -{=) INTERROGATIVE-TYPE)

((INTERROGATIVE-TYPE -[) wh-
     (Wh ^ Finite))

((wh- -{=) WH-FUNCTION)
```

```
((WH-FUNCTION -[ ) wh-subject
    (Wh / Subject) (% ^ Subject) (Finite ^ %))

((WH-FUNCTION -[ ) wh-other
    (Wh / Residual)
    (Finite ^ Subject)
    (% ^ Finite)
    (Subject ^ %))

((INTERROGATIVE-TYPE -[ ) yes/no
    (Finite ^ Subject) (% ^ Finite) (Subject ^ %))

((wh- -{=) THEME-MARKING-WH-)

((THEME-MARKING-WH- -[ ) unmarked-wh-theme
    (Topical / Wh))

((yes/no --) THEME-MARKING-YES/NO)

((THEME-MARKING-YES/NO -[ ) unmarked-yes/no-theme
    (Topical / Subject)
    (Theme (Interpersonal))
    (Interpersonal / Finite))

((MOOD-TYPE -[ ) imperative
    (Mood (Subject)) (Process : !stem))

((finite -{=) SUBJECT-PERSON)

((SUBJECT-PERSON -[ ) interactant-subject)

((indicative
  interactant-subject
  =}-) INDICATIVE-INTERACTANT-SUBJECT)

((INDICATIVE-INTERACTANT-SUBJECT -[ ) speaker-subject
    (Subject = I)
    (Finite : !first-person)
    (Finite : !v-singular))

((INDICATIVE-INTERACTANT-SUBJECT -[ ) addressee-subject
    (Subject = you) (Finite : !second-person))

((INDICATIVE-INTERACTANT-SUBJECT -[ ) speaker-plus-subject
    (Subject = we)
    (Finite : !first-person)
    (Finite : !v-plural))

((SUBJECT-PERSON -[ ) other-subject)

((other-subject -{=) INDICATIVE-OTHER-SUBJECT-NUMBER)
```

```
((INDICATIVE-OTHER-SUBJECT-NUMBER -[) mass-subject
     (Subject : mass)
     (Finite : !third-person)
     (Finite : !v-singular))

((INDICATIVE-OTHER-SUBJECT-NUMBER -[) singular-subject
     (Subject : singular)
     (Finite : !third-person)
     (Finite : !v-singular))

((INDICATIVE-OTHER-SUBJECT-NUMBER -[) plural-subject
     (Subject : plural)
     (Finite : !third-person)
     (Finite : !v-plural))

((imperative
  interactant-subject
  =}-) IMPERATIVE-INTERACTANT-SUBJECT)

((IMPERATIVE-INTERACTANT-SUBJECT -[) oblative
     (Subject = |let me|))

((IMPERATIVE-INTERACTANT-SUBJECT -[) jussive)

((IMPERATIVE-INTERACTANT-SUBJECT -[) suggestive
     (Subject = |let's|))

((jussive -{=) IMPERATIVE-SUBJECT-PRESUMPTION)

((indicative other-subject =}-) INDICATIVE-OTHER-SUBJECT)

((INDICATIVE-OTHER-SUBJECT -[) pronominal-subject
     (Subject : pronoun)
     (Subject<Head : !subjective)
     (Subject<Head : !third))

((INDICATIVE-OTHER-SUBJECT -[) nominal-subject)

((IMPERATIVE-SUBJECT-PRESUMPTION
  -[) imperative-subject-implicit)

((IMPERATIVE-SUBJECT-PRESUMPTION
  -[) imperative-subject-explicit
     (Subject = you))

((other-subject imperative =}-) IMPERATIVE-OTHER-SUBJECT)

((IMPERATIVE-OTHER-SUBJECT -[) proper-subject
     (Subject : Noun-head) (Subject<Head : !proper))

((IMPERATIVE-OTHER-SUBJECT -[) common-subject
     (Subject : Noun-head) (Subject<Head : !common))

((imperative -{=) IMPERATIVE-TAG)
```

```
((IMPERATIVE-TAG -[) imperative-untagged
     (Residue ^ #))

((IMPERATIVE-TAG -[) imperative-tagged
     (Residue ^ Moodtag)
     (Moodtag ^ #)
     (Moodtag (Tagfinite))
     (Moodtag (Tagsubject))
     (Tagfinite ^ Tagsubject)
     (% ^ Tagfinite)
     (Tagsubject ^ %))

((imperative-tagged oblative =}-) oblative-tagged
     (Tagfinite = shall) (Tagsubject = I))

((imperative-tagged jussive =}-) jussive-tagged
     (Tagfinite = will) (Tagsubject = you))

((imperative-tagged suggestive =}-) suggestive-tagged
     (Tagfinite = shall) (Tagsubject = we))

((finite -{=) POLARITY)

((POLARITY -[) positive)

((POLARITY -[) negative)

((positive -{=) POLARITY-MARKING-POSITIVE)

((POLARITY-MARKING-POSITIVE -[) unmarked-positive)

((POLARITY-MARKING-POSITIVE -[) marked-positive)

((negative -{=) POLARITY-MARKING-NEGATIVE)

((POLARITY-MARKING-NEGATIVE -[) unmarked-negative)

((POLARITY-MARKING-NEGATIVE -[) marked-negative)

((clause -{=) AGENCY)

((AGENCY -[) middle)

((AGENCY -[) effective)

((clause -{=) RESIDUALITY)

((RESIDUALITY -[) non-residual)

((RESIDUALITY -[) residual)

((declarative -{=) ATTITUDE)

((ATTITUDE -[) non-attitudinal)
```

```
((ATTITUDE -[ ) attitudinal)

((attitudinal --) INTERPERSONAL-THEME)

((INTERPERSONAL-THEME -[ ) no-interpersonal-theme)

((INTERPERSONAL-THEME -[ ) interpersonal-theme
     (Theme (Interpersonal))
     (Interpersonal / Attitude)
     (Interpersonal ^ Topical))

((middle -{=) RANGE)

((RANGE -[ ) non-ranged
     (Medium / Subject))

((RANGE -[ ) ranged)

((effective -{=) BENEFACTION)

((BENEFACTION -[ ) non-benefactive)

((BENEFACTION -[ ) benefactive)

((clause -{=) PROCESS-TYPE)

((PROCESS-TYPE -[ ) material)

((middle material =}-) HAPPENING)

((HAPPENING -[ ) behavioural
     (Actor / Medium)
     (Actor : conscious)
     (Process : behaviour))

((HAPPENING -[ ) eventive
     (Actor / Medium)
     (Actor : non-conscious)
     (Process : event))

((effective material =}-) DOING)

((DOING -[ ) creative
     (Actor / Agent) (Goal / Medium))

((DOING -[ ) dispositive
     (Actor / Agent) (Goal / Medium))

((PROCESS-TYPE -[ ) mental)

((mental -{=) SENSING)

((SENSING -[ ) perception)
```

```
((SENSING -[) reaction)

((SENSING -[) cognition)

((PROCESS-TYPE -[) verbal)

((verbal non-ranged =}-) SAYING)

((SAYING -[) indicating
     (Beta / Residual) (Beta : finite))

((indicating --) INDICATING)

((INDICATING -[) declaring
     (Beta : declarative))

((INDICATING -[) interrogating
     (Beta : interrogative))

((SAYING -[) imperating)

((PROCESS-TYPE -[) relational)

((middle relational =}-) TYPE-OF-BEING)

((TYPE-OF-BEING -[) ascriptive
     (Be-er / Carrier)
     (Carrier / Subject)
     (Attribute / Range))

((TYPE-OF-BEING -[) existential
     (Be-er / Existent)
     (Existent / Range)
     (-There- / Subject)
     (Process : !-be-))

((relational effective =}-) equative
     (Be-er / Identified))

((ascriptive equative ]-) RELATION-TYPE)

((RELATION-TYPE -[) intensive
     (Process : !-be-))

((RELATION-TYPE -[) circumstantial
     (Process : !-be-at-))

((RELATION-TYPE -[) possessive
     (Process : !-have-))

((middle intensive =}-) ATTRIBUTE-STATUS)
```

```
((ATTRIBUTE-STATUS -[) property-attribute
     (Value / Attribute)
     (Attribute : nominal-group-epithet-head))

((ATTRIBUTE-STATUS -[) class-attribute
     (Value / Attribute))

((effective intensive =}-) IDENTIFICATION-DIRECTION)

((IDENTIFICATION-DIRECTION -[) decoding
     (Identified / Token) (Identifier / Value))

((IDENTIFICATION-DIRECTION -[) encoding
     (Idintified / Value) (Identifier / Token))

((middle circumstantial =}-) CIRCUMSTANCE-AS-ATTRIBUTE)

((CIRCUMSTANCE-AS-ATTRIBUTE -[) circumstantial-ascription
     (Circumstance / Process))

((CIRCUMSTANCE-AS-ATTRIBUTE -[) circumstantial-attribute
     (Circumstance / Attribute))

((effective circumstantial =}-) CIRCUMSTANCE-AS-IDENTITY)

((CIRCUMSTANCE-AS-IDENTITY -[) circumstantial-equation
     (Circumstance / Process))

((CIRCUMSTANCE-AS-IDENTITY -[) circumstantial-identity
     (Circumstance / Identified)
     (Circumstance2 / Identifier))

((circumstantial -{=) CIRC-TYPE)

((CIRC-TYPE -[) relational-extent
     (Circumstance / Extent))

((CIRC-TYPE -[) relational-location
     (Circumstance / Locative))

((CIRC-TYPE -[) relational-cause
     (Circumstance / Cause))

((CIRC-TYPE -[) relational-manner
     (Circumstance / Manner))

((CIRC-TYPE -[) relational-accompaniment
     (Circumstance / Accompaniment))

((CIRC-TYPE -[) relational-matter
     (Circumstance / Matter))

((CIRC-TYPE -[) relational-role
     (Circumstance / Role))
```

```
((middle possessive =}-) POSS-AT)

((POSS-AT -[) possessive-ascription
     (Possession / Process))

((POSS-AT -[) possessive-attribute
     (Possession / Attribute))

((effective possessive =}-) POSS-ID)

((POSS-ID -[) possessive-equation
     (Possession / Process))

((POSS-ID -[) possessive-identity
     (Possession / Identified) (Possession2 / Identifier))

((indicative -{=) DEICTICITY)

((DEICTICITY -[) modal
     (Modal / Finite)
     (Residue (Modalstem))
     (% ^ Modalstem))

((DEICTICITY -[) temporal)

((temporal --) PRIMARY-TENSE)

((PRIMARY-TENSE -[) future
     (Future / Finite)
     (Future : !will)
     (Residue (Futstem))
     (% ^ Futstem))

((PRIMARY-TENSE -[) present
     (Finite : !present))

((PRIMARY-TENSE -[) past
     (Finite : !past))

((clause -{=) EXTENT)

((EXTENT -[) non-extent)

((EXTENT -[) extent
     (Extent / Residual))

((extent --) EXTENT-TYPE)

((EXTENT-TYPE -[) duration
     (Extent / Duration))

((EXTENT-TYPE -[) distance
     (Extent / Distance))
```

```
((clause -{=) TIME)

((TIME -[) non-time)

((TIME -[) time)

((time -[) non-textual-time
     (Residue (Temporal))
     (Temporal : prep-phrase)
     (Temporal ^ %))

((time -[) textual-time)

((clause -{=) PLACE)

((PLACE -[) non-place)

((PLACE -[) place
     (Residue (Spatial)) (Spatial : prep-phrase))

((clause -{=) CAUSE)

((CAUSE -[) non-cause)

((CAUSE -[) cause)

((cause --) CAUSE-TYPE)

((CAUSE-TYPE -[) reason)

((reason -[) non-textual-reason
     (Residual / Reason))

((reason -[) textual-reason)

((CAUSE-TYPE -[) purpose
     (Cause / Purpose) (Residual / Cause))

((CAUSE-TYPE -[) behalf
     (Cause / Behalf) (Residual / Cause))

((clause -{=) ACCOMPANIMENT)

((ACCOMPANIMENT -[) non-accompaniment)

((ACCOMPANIMENT -[) accompaniment
     (Accompaniment / Residual))

((clause -{=) MATTER)

((MATTER -[) non-matter)

((MATTER -[) matter
     (Matter / Residual))
```

```
((clause -{=) ROLE)

((ROLE -[) non-role)

((ROLE -[) role
     (Role / Residual))

((declarative -{=) THEME-MARKING-DECLARATIVE)

((THEME-MARKING-DECLARATIVE -[) unmarked-declarative-theme
     (Topical / Subject))

((imperative -{=) THEME-MARKING-IMPERATIVE)

((THEME-MARKING-IMPERATIVE -[) unmarked-imperative-theme)

((ranged -{=) RANGE-VOICE)

((RANGE-VOICE -[) range-operative
     (Medium / Subject) (Range / Residual))

((RANGE-VOICE -[) range-receptive
     (Range / Subject) (Lexverb : !en))

((range-receptive =}-) RANGE-MEDIATION)

((RANGE-MEDIATION -[) non-mediated)

((RANGE-MEDIATION -[) mediated
     (Residue (Medmarker))
     (Medmarker = by)
     (Medium / Residual)
     (Lexverb ^ Medmarker)
     (Medmarker ^ Medium))

((clause -{=) SECONDARY-TENSE-I)

((SECONDARY-TENSE-I -[) non-past-in)

((SECONDARY-TENSE-I -[) past-in
     (Residue (En)))

((clause -{=) SECONDARY-TENSE-II)

((SECONDARY-TENSE-II -[) non-present-in)

((SECONDARY-TENSE-II -[) present-in
     (Residue (Ing)))

((effective -{=) EFFECTIVE-VOICE)

((EFFECTIVE-VOICE -[) operative
     (Agent / Subject) (Medium / Residual))
```

```
((operative benefactive =}-) BENEFACTIVE-CULMINATION-I)

((BENEFACTIVE-CULMINATION-I -[) ben-med
     (Lexverb ^ Beneficiary) (Beneficiary ^ Medium))

((BENEFACTIVE-CULMINATION-I -[) med-ben
     (Lexverb ^ Medium)
     (Medium ^ Benmarker)
     (Benmarker ^ Beneficiary)
     (Benmarker = to))

((EFFECTIVE-VOICE -[) receptive
     (Lexverb : !en))

((benefactive receptive =}-) BENEFACTIVE-VOICE)

((BENEFACTIVE-VOICE -[) medioreceptive
     (Medium / Subject))

((BENEFACTIVE-VOICE -[) benereceptive
     (Beneficiary / Subject))

((medioreceptive agentive =}-) BENEFACTIVE-CULMINATION-III)

((BENEFACTIVE-CULMINATION-III -[) ben-ag
     (Benmarker = to)
     (Benmarker ^ Beneficiary)
     (Lexverb ^ Benmarker)
     (Beneficiary ^ Agentmarker))

((BENEFACTIVE-CULMINATION-III -[) ag-ben
     (Benmarker = to)
     (Benmarker ^ Beneficiary)
     (Lexverb ^ Agentmarker)
     (Agent ^ Beneficiary))

((benereceptive agentive =}-) BENEFACTIVE-CULMINATION-II)

((BENEFACTIVE-CULMINATION-II -[) med-ag
     (Lexverb ^ Medium) (Medium ^ Agentmarker))

((BENEFACTIVE-CULMINATION-II -[) ag-med
     (Lexverb ^ Agentmarker) (Agent ^ Medium))

((receptive -{=) AGENTIVITY)

((AGENTIVITY -[) non-agentive)

((AGENTIVITY -[) agentive
     (Residue (Agentmarker))
     (Agent / Residual)
     (Agentmarker = by))

((clause -{=) TEXTUAL-THEME)
```

```
((TEXTUAL-THEME -[) non-textual-theme)

((TEXTUAL-THEME -[) textual-theme
     (Theme (Textual)) (% ^ Textual) (Textual / Conjunct))

((indicative positive =}-) positive-finite
     (Finite : !positive))

((indicative negative =}-) negative-finite
     (Finite : !negative))

((imperative negative =}-) negimperative-finite
     (Mood (Finite))
     (% ^ Finite)
     (Finite / Dont)
     (Finite / Topical))

((negimperative-finite
  imperative-subject-implicit
  =}-) neg-imp-subj
     (Finite ^ %))

((declarative
  unmarked-positive
  (pronominal-subject interactant-subject ]-)
  =}-) reduced-posfinite
     (Finite : !reduced))

((indicative
  (interrogative marked-positive nominal-subject ]-)
  =}-) nonreduced-posfinite
     (Finite : !nonreduced))

((marked-positive imperative =}-) pos-do-finite
     (Finite : !do) (Finite ^ Subject))

((indicative unmarked-negative =}-) reduced-negfinite
     (Finite : !reduced))

((imperative marked-negative =}-) do-not-finite
     (Dont = |do not|))

((indicative marked-negative =}-) nonreduced-negfinite
     (Finite : !nonreduced))

((imperative unmarked-negative =}-) dont-finite
     (Dont = |don't|))

((benereceptive non-agentive =}-) ben-non-ag
     (Residual / Medium) (Lexverb ^ Residual))

((non-benefactive receptive =}-) non-bene-reception
     (Medium / Subject))
```

```
((non-bene-reception agentive =}-) ag-non-bene-reception
     (Lexverb ^ Agentmarker))

((operative range-operative non-ranged ]-) active-process)

((receptive range-receptive ]-) passive-process
     (Pass ^ Lexverb))

((past-in present-in =}-) enprog
     (Prog = been) (En / Prog))

((modal past-in =}-) modalstemperf
     (Perf = have) (Modalstem / Perf))

((modal non-past-in present-in =}-) modalstemprog
     (Prog = be) (Modalstem / Prog))

((future past-in =}-) futstemperf
     (Perf = have) (Futstem / Perf))

((future non-past-in present-in =}-) futstemprog
     (Prog = be) (Futstem / Prog))

((future
  passive-process
  non-past-in
  non-present-in
  =}-) futstempass
     (Pass = be) (Futstem / Pass))

((active-process
  non-past-in
  non-present-in
  future
  =}-) futstemlexverb
     (Lexverb : !stem) (Futstem / Lexverb))

(((past present ]-) past-in =}-) finiteperf
     (Finite / Perf) (% ^ En))

(((past present ]-) non-past-in present-in =}-) finiteprog
     (Finite / Prog) (% ^ Ing))

((passive-process
  (past present ]-)
  non-past-in
  non-present-in
  =}-) finitepass
     (Finite / Pass) (% ^ Lexverb))

((passive-process past-in non-present-in =}-) enpass
     (Pass = been) (En / Pass))
```

```
((active-process past-in non-present-in =}-) enlexverb
     (Lexverb : !en) (En / Lexverb))

((present-in passive-process =}-) ingpass
     (Pass = being) (Ing / Pass))

((active-process present-in =}-) inglexverb
     (Lexverb : !ing) (Ing / Lexverb))

((modal
  non-past-in
  non-present-in
  passive-process
  =}-) modalstempass
     (Pass = be) (Modalstem / Pass))

((modal
  non-past-in
  non-present-in
  active-process
  =}-) modalstemlexverb
     (Lexverb : !stem) (Modalstem / Lexverb))

(((finiteprog finitepass ]-)
  present
  speaker-subject
  =}-) am
     (Finite : !am))

(((finiteprog finitepass ]-)
  present
  (addressee-subject
   speaker-plus-subject
   plural-subject
   ]-)
  =}-) are
     (Finite : !are))

(((finiteprog finitepass ]-)
  present
  (singular-subject mass-subject ]-)
  =}-) is
     (Finite : !is))

(((finiteprog finitepass ]-)
  past
  (singular-subject mass-subject speaker-subject ]-)
  =}-) was
     (Finite : !was))
```

```
(((finiteprog finitepass ]-)
  past
  (addressee-subject
   speaker-plus-subject
   plural-subject
   ]-)
  =}-) were
     (Finite : !were))

((finiteperf
  present
  (plural-subject
   speaker-subject
   addressee-subject
   speaker-plus-subject
   ]-)
  =}-) have
     (Finite : !have))

((finiteperf
  present
  (singular-subject mass-subject ]-)
  =}-) has
     (Finite : !has))

((finiteperf past =}-) had
     (Finite : !had))

((positive (declarative imperative ]-) =}-) assertive)

((negative interrogative ]-) non-assertive)

(((past present imperative ]-)
  active-process
  non-past-in
  non-present-in
  =}-) not-auxed)

((circumstantial-ascription
  possessive-ascription
  active-process
  material
  mental
  verbal
  ]-) do-needing-verbs)

((non-assertive not-auxed do-needing-verbs =}-) do-finite
     (Lexverb : !stem) (% ^ Lexverb))

((modal past-in =}-) perf-en
     (Perf ^ En))

(((future modal past-in ]-) present-in =}-) prog-ing
     (Prog ^ Ing))
```

```
((past do-finite =}-) did
     (Finite : !did))

((present
  do-finite
  (plural-subject
   addressee-subject
   speaker-plus-subject
   speaker-subject
   ]-)
  =}-) do
     (Finite : !do))

((present
  do-finite
  (mass-subject singular-subject ]-)
  =}-) does
     (Finite : !does))

((indicative
  (imperative interactant-subject =}-)
  subject-specified
  ]-) subject-inserted)

(((subject-inserted middle =}-)
  effective
  ]-) medium-inserted)

((medium-inserted mental =}-) senser-inserted
     (Medium / Senser))

((medium-inserted verbal =}-) sayer-inserted
     (Medium / Sayer))

((medium-inserted relational =}-) be-er-inserted
     (Medium / Be-er))

((imperative-subject-implicit
  unmarked-imperative-theme
  unmarked-positive
  =}-) topical-predicator
     (Topical / Process) (Process : stem))

((unmarked-imperative-theme
  (imperative-subject-explicit suggestive oblative ]-)
  =}-) topical-impsubject
     (Topical / Subject))

((topical-predicator
  (negative marked-positive ]-)
  =}-) neg-predicator
     (% ^ Finite))
```

```
((clause -{=) topical-inserted
     (Theme (Topical)) (Topical ^ %))

((non-textual-theme
  (marked-yes/no-theme
   no-interpersonal-theme
   non-attitudinal
   wh-
   imperative
   ]-)
  =}-) topical-only
     (% ^ Topical))

((non-textual-theme
  (interpersonal-theme unmarked-yes/no-theme ]-)
  =}-) int-topical
     (% ^ Interpersonal))

((textual-theme
  (marked-yes/no-theme
   no-interpersonal-theme
   non-attitudinal
   wh-
   imperative
   ]-)
  =}-) text-topic
     (Textual ^ Topical))

((textual-theme
  (interpersonal-theme unmarked-yes/no-theme ]-)
  =}-) text-int
     (Textual ^ Interpersonal))

((residual wh-other =}-) fronted-residual
     (Lexverb ^ %))

((residual
  (wh-subject yes/no declarative imperative non-finite ]-)
  =}-) unmarked-residual
     (Residue (Residual)))

((unmarked-residual processfinite =}-) res-first
     (% ^ Residual))

((unmarked-residual place =}-) res-place
     (Residual ^ Spatial))

((unmarked-residual
  non-place
  non-textual-time
  =}-) res-time
     (Residual ^ Temporal))
```

```
((unmarked-residual
  non-place
  (non-time textual-time ]-)
  =}-) residual-final
     (Residual ^ %))

((agentive wh-other =}-) agentive-wh-other
     (Agentmarker ^ %))

((agentive unmarked-residual =}-) unmarked-agentive
     (Agentmarker ^ Residual))

((not-auxed assertive =}-) processfinite
     (% ^ Residual) (Process / Finite))

((interrogative
  future
  passive-process
  negative
  modal
  past-in
  present-in
  ]-) processlexverb
     (Residue (Lexverb)) (Process / Lexverb))

((processlexverb
  active-process
  unmarked-residual
  =}-) unmarked-operative
     (Lexverb ^ Residual))

((non-residual (non-place textual-time ]-) =}-) lex-final
     (Lexverb ^ %))

((non-residual place =}-) lex-place
     (Lexverb ^ Spatial))

((place (non-time textual-time ]-) =}-) place-final
     (Spatial ^ %))

((non-residual non-place non-textual-time =}-) lex-time
     (Lexverb ^ Temporal))

((place non-textual-time =}-) place-time
     (Spatial ^ Temporal))

((unmarked-positive imperative =}-) unmarked-imppos
     (Subject ^ Finite) (Finite ^ %))
```

2. The nominal-group network

This network was taken from (Halliday, 1976a, p. 131).

```
(nil nominal-group
     (Head ^ #))

((nominal-group -{=) CLASS-AT-HEAD)

((CLASS-AT-HEAD -[) nominal)

((nominal -[) pronoun
     (# ^ Head))

((pronoun -[) non-personal)

((non-personal -[) -it-
     (Head = it))

((non-personal -[) -there-
     (Head = there))

((pronoun -[) personal
     (Head : !personal))

((nominal -[) noun)

((nominal -[) substitute
     (Head = one))

((noun substitute ]-) QUANTIFICATION)

((QUANTIFICATION -[) non-quantified)

((QUANTIFICATION -[) quantified)

((CLASS-AT-HEAD -[) non-nominal)

((non-nominal -[) determiner-head
     (Deictic / Head) (# ^ Deictic))

((non-nominal -[) quantifier-head)

((quantifier-head -[) superlative-quant)

((quantifier-head -[) quant)

((quantifier-head -[) comparative-quant)

((quantifier-head -[) superlative-adj)

((noun
  substitute
  superlative-quant
  quant
  comparative-quant
  superlative-adj
  ]-) DETERMINATION)
```

```
((DETERMINATION -[) non-determined)

((DETERMINATION -[) determined
     (# ^ Deictic))

((non-nominal -[) non-superlative-adj-head)

((non-superlative-adj-head -[) comparative)

((non-superlative-adj-head -[) intensified)

((non-superlative-adj-head -[) unmarked)

((nominal-group -{=) POSSESSION)

((POSSESSION -[) non-possessive-nom)

((POSSESSION -[) possessive-nom)

((nominal-group -{=) NUMBER)

((NUMBER -[) count)

((count -[) singular)

((count -[) plural)

((NUMBER -[) mass)

((personal non-possessive-nom -<>) non-pers-pos)

((personal possessive-nom -<>) personal-possessive
     (Head : !possessive))

(((determiner-head pronoun ]-)
  (singular mass ]-)
  =}-) sing-pro
     (Head : !singular-pronoun))

(((determiner-head pronoun ]-) plural =}-) plur-pro
     (Head : !plural-pronoun))

(((noun substitute ]-) (singular mass ]-) =}-) sing-noun
     (Head : !singular))

(((noun substitute ]-) plural =}-) plur-noun
     (Head : !plural))

((pronoun possessive-nom =}-) poss-pro
     (Head : !possessive-pronoun))

(((noun substitute ]-) possessive-nom =}-) poss-noun
     (Head : !possessive))
```

```
(((noun substitute ]-)
  non-possessive-nom
  =}-) non-poss-noun
     (Head : !non-possessive))

((determined non-quantified nominal =}-) det-non-quant
     (Deictic ^ Head))

((non-determined non-quantified =}-) head-only
     (# ^ Head))
```

3. The determiner network

This network was taken from (Halliday, 1976a, pp. 132-133), with slight extensions and modifications from (Quirk, Greenbaum, Leech and Svartvik, 1973, 4.122). The "selective" system was rearranged to be compatible with (Winograd, 1983, p. 537).

```
((determined determiner-head ]-) DETERMINER)

((DETERMINER -[ ) specific)

((specific -[ ) non-selective
     (Deictic = the))

((specific -[ ) selective)

((selective -[ ) pronominal-det)

((pronominal-det -[ ) interrogative-det)

((interrogative-det -[ ) wh-det
     (Deictic : !wh))

((interrogative-det -[ ) wh-ever-det
     (Deictic : !wh-ever))

((pronominal-det -[ ) non-interrogative-det)

((pronominal-det -[ ) demonstrative)

(((demonstrative interrogative-det -<>) -[ ) which
     (Deictic : !which))

(((demonstrative interrogative-det -<>) -[ ) what
     (Deictic : !what))

(((demonstrative non-interrogative-det -<>) -[ ) near
     (Deictic : !near))
```

```
(((demonstrative non-interrogative-det -<>) -[] far
     (Deictic : !far))

((pronominal-det -[] possessive-det
     (Deictic : !possessive-determiner))

((selective -[] nominal-det
     (Deictic : possessive-nom))

((DETERMINER -[] non-specific)

((non-specific -[] total)

((total -[] positive-det)

(((positive-det singular -<>) -[] individual
     (Deictic = each))

(((positive-det singular -<>) -[] group
     (Deictic = every))

(((positive-det (mass plural ]-) -<>) -[] exactly-two
     (Deictic = both))

(((positive-det (mass plural ]-) -<>) -[] >two
     (Deictic = all))

((total -[] negative-det)

((negative-det -[] quasi-negative)

((negative-det -[] full-negative)

((non-specific -[] partial)

((partial -[] part-non-selective)

(((part-non-selective singular -<>) -[] additional
     (Deictic = another))

(((part-non-selective singular -<>) -[] non-add
     (Deictic = a))

(((part-non-selective (mass plural ]-) -<>) -[] sufficient
     (Deictic = enough))

(((part-non-selective (mass plural ]-) -<>)
  -[] non-sufficient
     (Deictic = some))

((partial -[] part-selective)

((part-selective -[] restricted)
```

```
((part-selective -[ ) unrestricted)

(((singular mass ]-)
  non-interrogative-det
  demonstrative
  =}-) sing-pro-det
     (Deictic : !singular-pronoun))

((plural
  non-interrogative-det
  demonstrative
  =}-) plur-pro-det
     (Deictic : !plural-pronoun))

(((quasi-negative unrestricted ]-) singular -<>) "either"
     (Deictic = either))

(((quasi-negative unrestricted ]-)
  (mass plural ]-)
  -<>) "any"
     (Deictic = any))

((full-negative singular -<>) "neither"
     (Deictic = neither))

((full-negative (mass plural ]-) -<>) "no"
     (Deictic = no))

((restricted singular -<>) "one"
     (Deictic = "one"))

((restricted (mass plural ]-) -<>) "some"
     (Deictic = some))
```

4. The quantifier network

This network was taken from (Halliday, 1976a, pp. 134-135).

```
((superlative-quant -[ ) numerical)

((superlative-quant -[ ) non-numerical)

((quantifier -[ ) cardinal)

((cardinal -[ ) integer)

((cardinal -[ ) fraction)

((quantifer -[ ) indefinite)

((indefinite -[ ) quantitative)
```

```
((quantitative -[ ) multal1)

((quantitative -[ ) neutral1
     (Numerative = several))

((quantitative -[ ) paucal1)

((indefinite -[ ) partitive)

((partitive -[ ) multal2
     (Numerative = |lots of|))

((partitive -[ ) neutral2)

((partitive -[ ) paucal2)

((comparative-quant -[ ) multal3
     (Numerative = more))

((comparative-quant -[ ) paucal3)

((superlative-adj -[ ) multal4
     (Numerative = most))

((superlative-adj -[ ) paucal4)

((integer singular -<>) one
     (Numerative = one))

((multal1 mass -<>) much
     (Numerative = much))

((multal1 plural -<>) many
     (Numerative = many))

((paucal1 mass -<>) little
     (Numerative = little))

((paucal1 plural -<>) few
     (Numerative = few))

((neutral2 mass -<>) a-number-of
     (Numerative = |a number of|))

((neutral2 plural -<>) a-certain-amount-of
     (Numerative = |a certain amount of|))

((paucal2 mass -<>) a-little
     (Numerative = |a little|))

((paucal2 plural -<>) a-few
     (Numerative = |a few|))
```

```
((paucal3 mass -<>) less
     (Numerative = less))

((paucal3 plural -<>) fewer
     (Numerative = fewer))

((paucal4 mass -<>) least
     (Numerative = least))

((paucal4 plural -<>) fewest
     (Numerative = fewest))
```

5. The prepositional-phrase network

This is described in (Halliday, 1985, pp. 189-190).

```
(nil prep-phrase
     (Range : nominal-group) (Range ^ #))

((prep-phrase -[) unmarked-prep-phrase
     (# ^ Minor-process)
     (Minor-process ^ Range)
     (Minor-process : !preposition))

((prep-phrase -[) merged
     (# ^ Range))
```

6. The verb network

This is an integrated verb and auxiliary network, very loosely fashioned after (Winograd, 1983, p. 534).

```
(nil !verb)

((!verb -{=) VERB-TYPE)

((VERB-TYPE -[) !aux
     (# ^ !Aux))

((!aux -{=) AUX-POLARITY)

((AUX-POLARITY -[) !negative)

((AUX-POLARITY -[) !positive
     (!Aux ^ #))

((!aux -{=) AUX-MODALITY)

((AUX-MODALITY -[) !modal-aux)
```

```
((!modal-aux -[] !could)

((!modal-aux -[] !would)

((!modal-aux -[] !should)

((!modal-aux -[] !must)

((!modal-aux -[] !can)

((!modal-aux -[] !might)

((!modal-aux -[] !will)

((AUX-MODALITY -[] !non-modal-aux)

((!non-modal-aux -[] !be-aux)

((!be-aux -[] !am)

((!be-aux -[] !are)

((!be-aux -[] !is)

((!be-aux -[] !was)

((!be-aux -[] !were)

((!non-modal-aux -[] !do-aux)

((!do-aux -[] !do)

((!do-aux -[] !did)

((!do-aux -[] !does)

((!non-modal-aux -[] !have-aux)

((!have-aux -[] !have)

((!have-aux -[] !had)

((!have-aux -[] !has)

((VERB-TYPE -[] !non-aux
     (# ^ !Verb) (!Verb ^ #))

((!non-aux -[] !achieve)

((!non-aux -[] !fasten)

((!non-aux -[] !pour)

((!non-aux -[] !begin)
```

```
((!non-aux -[ ) !lay)
((!non-aux -[ ) !finish)
((!non-aux -[ ) !sand)
((!non-aux -[ ) !varnish)
((!non-aux -[ ) !build)
((!non-aux -[ ) !give)
((!non-aux -[ ) !go)
((!non-aux -[ ) !-do-)
((!non-aux -[ ) !-be-)
((!non-aux -[ ) !-have-)
((!non-aux -[ ) !put)
((!non-aux -[ ) !support)
((!non-aux -[ ) !ask)
((!non-aux -[ ) !take)
((!non-aux -[ ) !smack)
((!aux !non-aux ]-) VTENSE)
((VTENSE -[ ) !past)
((VTENSE -[ ) !present)
((!past !present ]-) VNUMBER)
((VNUMBER -[ ) !v-singular)
((VNUMBER -[ ) !v-plural)
((!past !present ]-) VPERSON)
((VPERSON -[ ) !first-person)
((VPERSON -[ ) !second-person)
((VPERSON -[ ) !third-person)
((VTENSE -[ ) !infinitive)
((VTENSE -[ ) !stem)
```

```
((VTENSE -[ ) !ing)

((VTENSE -[ ) !en)

((!verb -{=) VERB-REDUCTION)

((VERB-REDUCTION -[ ) !reduced)

((VERB-REDUCTION -[ ) !nonreduced)

((!negative !nonreduced =}-) !negnonreduced)

((!aux !negative !reduced =}-) !auxnt
    (!Aux ^ #))

((!aux !negnonreduced =}-) !aux-not
    (!Neg = not) (!Aux ^ !Neg) (!Neg ^ #))

((!am ((!positive !nonreduced =}-) !negative ]-) =}-) "am"
    (!Aux = am))

((!am !positive !reduced =}-) "m"
    (!Aux = |'m|))

((!are !nonreduced =}-) "are"
    (!Aux = are))

((!are !positive !reduced =}-) "re"
    (!Aux = |'re|))

((!are !negative !reduced =}-) "arent"
    (!Aux = |aren't|))

((!is !nonreduced =}-) "is"
    (!Aux = is))

(((!is !has ]-) !positive !reduced =}-) "s"
    (!Aux = |'s|))

((!is !negative !reduced =}-) "isnt"
    (!Aux = |isn't|))

((!was (!positive !negnonreduced ]-) =}-) "was"
    (!Aux = was))

((!was !negative !reduced =}-) "wasnt"
    (!Aux = |wasn't|))

((!were (!positive !negnonreduced ]-) =}-) "were"
    (!Aux = were))

((!were !negative !reduced =}-) "werent"
    (!Aux = |weren't|))
```

```
((!will !nonreduced =}-) "will"
     (!Aux = will))

((!will !positive !reduced =}-) "ll"
     (!Aux = |'ll|))

((!will !negative !reduced =}-) "wont"
     (!Aux = |won't|))

((!have !nonreduced =}-) "have"
     (!Aux = have))

((!have !positive !reduced =}-) "ve"
     (!Aux = |'ve|))

((!have !negative !reduced =}-) "havent"
     (!Aux = |haven't|))

((!has !nonreduced =}-) "has"
     (!Aux = has))

((!has !negative !reduced =}-) "hasnt"
     (!Aux = |hasn't|))

((!had !nonreduced =}-) "had"
     (!Aux = had))

((!had !negative !reduced =}-) "hadnt"
     (!Aux = |hadn't|))

((!do (!positive !negnonreduced ]-) =}-) "do"
     (!Aux = do))

((!do !negative !reduced =}-) "dont"
     (!Aux = |don't|))

((!did (!positive !negnonreduced ]-) =}-) "did"
     (!Aux = did))

((!did !negative !reduced =}-) "didnt"
     (!Aux = |didn't|))

((!does (!positive !negnonreduced ]-) =}-) "does"
     (!Aux = does))

((!does !negative !reduced =}-) "doesnt"
     (!Aux = |doesn't|))

((!could !nonreduced =}-) "could"
     (!Aux = could))
```

```
(((!could !would !should !had ]-)
  !positive
  !reduced
  =}-) "d"
     (!Aux = |'d|))

((!could !negative !reduced =}-) "couldnt"
     (!Aux = |couldn't|))

((!would !nonreduced =}-) "would"
     (!Aux = would))

((!would !negative !reduced =}-) "wouldnt"
     (!Aux = |wouldn't|))

((!should !nonreduced =}-) "should"
     (!Aux = should))

((!should !negative !reduced =}-) "shouldnt"
     (!Aux = |shouldn't|))

((!must (!positive !negnonreduced ]-) =}-) "must"
     (!Aux = must))

((!must !negative !reduced =}-) "mustnt"
     (!Aux = |mustn't|))

((!can (!positive !negnonreduced ]-) =}-) "can"
     (!Aux = can))

((!can !negative !reduced =}-) "cant"
     (!Aux = |can't|))

((!might (!positive !negnonreduced ]-) =}-) "might"
     (!Aux = might))

((!might !negative !reduced =}-) "mightnt"
     (!Aux = |mightn't|))

((!present !third-person !v-singular =}-) !sing3)

((!past
  !first-person
  !second-person
  !v-plural
  ]-) !non-sing3)

((!achieve (!stem !non-sing3 ]-) =}-) "achieve"
     (!Verb = achieve))

((!achieve !sing3 =}-) "achieves"
     (!Verb = achieves))
```

```
((!achieve !infinitive =}-) "to-achieve"
     (!Verb = |to achieve|))

((!achieve (!en !past ]-) =}-) "achieved"
     (!Verb = achieved))

((!achieve !ing =}-) "achieving"
     (!Verb = achieving))

((!fasten (!stem !non-sing3 ]-) =}-) "fasten"
     (!Verb = fasten))

((!fasten !sing3 =}-) "fastens"
     (!Verb = fastens))

((!fasten !infinitive =}-) "to-fasten"
     (!Verb = |to fasten|))

((!fasten (!en !past ]-) =}-) "fastened"
     (!Verb = fastened))

((!fasten !ing =}-) "fastening"
     (!Verb = fastening))

((!pour (!stem !non-sing3 ]-) =}-) "pour"
     (!Verb = pour))

((!pour !sing3 =}-) "pours"
     (!Verb = pours))

((!pour !infinitive =}-) "to-pour"
     (!Verb = |to pour|))

((!pour (!en !past ]-) =}-) "poured"
     (!Verb = poured))

((!pour !ing =}-) "pouring"
     (!Verb = pouring))

((!begin (!stem !non-sing3 ]-) =}-) "begin"
     (!Verb = begin))

((!begin !sing3 =}-) "begins"
     (!Verb = begins))

((!begin !infinitive =}-) "to-begin"
     (!Verb = |to begin|))

((!begin (!en !past ]-) =}-) "began"
     (!Verb = began))

((!begin !ing =}-) "beginning"
     (!Verb = beginning))
```

```
((!finish (!stem !non-sing3 ]-) =}-) "finish"
     (!Verb = finish))

((!finish !sing3 =}-) "finishes"
     (!Verb = finishes))

((!finish !infinitive =}-) "to-finish"
     (!Verb = |to finish|))

((!finish (!en !past ]-) =}-) "finished"
     (!Verb = finished))

((!finish !ing =}-) "finishing"
     (!Verb = finishing))

((!lay (!stem !non-sing3 ]-) =}-) "lay"
     (!Verb = lay))

((!lay !sing3 =}-) "lays"
     (!Verb = lays))

((!lay !infinitive =}-) "to-lay"
     (!Verb = |to lay|))

((!lay (!en !past ]-) =}-) "laid"
     (!Verb = laid))

((!lay !ing =}-) "laying"
     (!Verb = laying))

((!sand (!stem !non-sing3 ]-) =}-) "sand"
     (!Verb = sand))

((!sand !sing3 =}-) "sands"
     (!Verb = sands))

((!sand !infinitive =}-) "to-sand"
     (!Verb = |to sand|))

((!sand (!en !past ]-) =}-) "sanded"
     (!Verb = sanded))

((!sand !ing =}-) "sanding"
     (!Verb = sanding))

((!varnish (!stem !non-sing3 ]-) =}-) "varnish"
     (!Verb = varnish))

((!varnish !sing3 =}-) "varnishes"
     (!Verb = varnishes))

((!varnish !infinitive =}-) "to-varnish"
     (!Verb = |to varnish|))
```

```
((!varnish (!en !past ]-) =}-) "varnished"
     (!Verb = varnished))

((!varnish !ing =}-) "varnishing"
     (!Verb = varnishing))

((!build (!stem !non-sing3 ]-) =}-) "build"
     (!Verb = build))

((!build !sing3 =}-) "builds"
     (!Verb = builds))

((!build !infinitive =}-) "to-build"
     (!Verb = |to build|))

((!build (!en !past ]-) =}-) "built"
     (!Verb = built))

((!build !ing =}-) "building"
     (!Verb = building))

((!give (!stem !non-sing3 ]-) =}-) "give"
     (!Verb = give))

((!give !sing3 =}-) "gives"
     (!Verb = gives))

((!give !infinitive =}-) "to-give"
     (!Verb = |to give|))

((!give !past =}-) "gave"
     (!Verb = gave))

((!give !ing =}-) "giving"
     (!Verb = giving))

((!give !en =}-) "given"
     (!Verb = given))

((!go (!stem !non-sing3 ]-) =}-) "go"
     (!Verb = go))

((!go !sing3 =}-) "goes"
     (!Verb = goes))

((!go !infinitive =}-) "to-go"
     (!Verb = |to go|))

((!go !past =}-) "went"
     (!Verb = went))

((!go !ing =}-) "going"
     (!Verb = going))
```

```
((!go !en =}-) "gone"
     (!Verb = gone))

((!-do- (!stem !non-sing3 ]-) =}-) "-do-"
     (!Verb = do))

((!-do- !sing3 =}-) "-do-es"
     (!Verb = does))

((!-do- !infinitive =}-) "to-do"
     (!Verb = |to do|))

((!-do- !past =}-) "-did-"
     (!Verb = did))

((!-do- !ing =}-) "-doing-"
     (!Verb = doing))

((!-do- !en =}-) "-done-"
     (!Verb = done))

((!-be- !stem =}-) "-be-"
     (!Verb = be))

((!-be- !infinitive =}-) "to-be"
     (!Verb = |to be|))

((!-be- !first-person !singular =}-) "-am-"
     (!Verb = am))

((!-be-
  !present
  (!plural (!second-person !singular =}-) ]-)
  =}-) "-are-"
     (!Verb = are))

((!-be- !sing3 !present =}-) "-is-"
     (!Verb = is))

((!-be-
  !past
  (!first-person !third-person ]-)
  !singular
  =}-) "-was-"
     (!Verb = was))

((!-be-
  !past
  ((!second-person !singular =}-)
   (!third-person !plural =}-)
   ]-)
  =}-) "-were-"
     (!Verb = were))
```

```
((!-be- !ing =}-) "-being-"
     (!Verb = being))

((!-be- !en =}-) "-been-"
     (!Verb = been))

((!-have- (!stem !non-sing3 ]-) =}-) "-have-"
     (!Verb = have))

((!-have- !sing3 =}-) "-has-"
     (!Verb = has))

((!-have- !infinitive =}-) "-to-have-"
     (!Verb = |to have|))

((!-have- (!en !past ]-) =}-) "-had-"
     (!Verb = had))

((!-have- !ing =}-) "-having-"
     (!Verb = having))

((!support (!stem !non-sing3 ]-) =}-) "support"
     (!Verb = support))

((!support !sing3 =}-) "supports"
     (!Verb = supports))

((!support !infinitive =}-) "to-support"
     (!Verb = |to support|))

((!support (!en !past ]-) =}-) "supported"
     (!Verb = supported))

((!support !ing =}-) "supporting"
     (!Verb = supporting))

((!smack (!stem !non-sing3 ]-) =}-) "smack"
     (!Verb = smack))

((!smack !sing3 =}-) "smacks"
     (!Verb = smacks))

((!smack !infinitive =}-) "to-smack"
     (!Verb = |to smack|))

((!smack (!en !past ]-) =}-) "smacked"
     (!Verb = smacked))

((!smack !ing =}-) "smacking"
     (!Verb = smacking))

((!put (!stem !non-sing3 !en !past ]-) =}-) "put"
     (!Verb = put))
```

```
((!put !sing3 =}-) "puts"
     (!Verb = puts))

((!put !infinitive =}-) "to-put"
     (!Verb = |to put|))

((!put !ing =}-) "putting"
     (!Verb = putting))

((!ask (!stem !non-sing3 ]-) =}-) "ask"
     (!Verb = ask))

((!ask !sing3 =}-) "asks"
     (!Verb = asks))

((!ask !infinitive =}-) "to-ask"
     (!Verb = |to ask|))

((!ask (!en !past ]-) =}-) "asked"
     (!Verb = asked))

((!ask !ing =}-) "asking"
     (!Verb = asking))

((!take (!stem !non-sing3 ]-) =}-) "take"
     (!Verb = take))

((!take !sing3 =}-) "takes"
     (!Verb = takes))

((!take !infinitive =}-) "to-take"
     (!Verb = |to take|))

((!take !en =}-) "taken"
     (!Verb = taken))

((!take !past =}-) "took"
     (!Verb = took))

((!take !ing =}-) "taking"
     (!Verb = taking))
```

7. The noun network

The important systems from this network were taken from (Winograd, 1983, p. 537).

```
(nil !noun
     (# ^ !Noun) (!Noun ^ #))

((!noun -[ ) !pronoun)
```

```
((!pronoun -[] !question)

((!question -[] !animate
     (!Noun = who))

((!question -[] !inanimate)

((!inanimate -[] !what
     (!Noun = what))

((!inanimate -[] !why
     (!Noun = why))

((!inanimate -[] !where
     (!Noun = where))

((!inanimate -[] !when
     (!Noun = when))

((!pronoun -[] !personal)

((!personal !question ]-) PRONOUN-CASE)

((PRONOUN-CASE -[] !subjective)

((PRONOUN-CASE -[] !objective)

((PRONOUN-CASE -[] !reflexive)

((PRONOUN-CASE -[] !possessive-pronoun)

((PRONOUN-CASE -[] !possessive-determiner)

((!personal -{=) PRONOUN-PERSON)

((PRONOUN-PERSON -[] !first)

((PRONOUN-PERSON -[] !second)

((PRONOUN-PERSON -[] !third)

((!pronoun -[] !demonstrative)

((!demonstrative !personal ]-) PRONOUN-NUMBER)

((PRONOUN-NUMBER -[] !singular-pronoun)

((!third !singular =}-) PRONOUN-GENDER)

((PRONOUN-GENDER -[] !feminine)

((PRONOUN-GENDER -[] !masculine)

((PRONOUN-GENDER -[] !neuter)
```

```
((PRONOUN-NUMBER -[ ) !plural-pronoun)
((!demonstrative -[ ) !near)
((!demonstrative -[ ) !far)
((!noun -[ ) !proper)
((!proper -[ ) !daddy)
((!proper -[ ) !mary)
((!proper -[ ) !scotland)
((!proper -[ ) !cwren)
((!proper -[ ) !taj-mahal)
((!proper -[ ) !table)
((!proper -[ ) !blockA)
((!proper -[ ) !blockB)
((!proper -[ ) !blockC)
((!proper -[ ) !blockD)
((!noun -[ ) !common)
((!proper !common ]-) NOUN-NUMBER)
((NOUN-NUMBER -[ ) !singular)
((NOUN-NUMBER -[ ) !plural)
(((!common !proper ]-) -[ ) !possessive)
(((!common !proper ]-) -[ ) !non-possessive)
((!common -[ ) !upstairs)
((!common -[ ) !headache)
((!common -[ ) !infection)
((!common -[ ) !fever)
((!common -[ ) !possibility)
((!common -[ ) !gazebo)
((!common -[ ) !sweet)
```

```
((!common -[ ) !floor)

((!common -[ ) !basement-floor)

((!common -[ ) !finished-flooring)

((!common -[ ) !flooring)

((!common -[ ) !painter)

((!common -[ ) !painting)

((!common -[ ) !carpentry)

((!common -[ ) !plasterer)

((!common -[ ) !plaster-board)

((!common -[ ) !plan)

((!common -[ ) !task-decorate)

((!singular-pronoun !first !subjective =}-) "I"
     (!Noun = I))

((!singular-pronoun !first !objective =}-) "me"
     (!Noun = me))

((!plural-pronoun !first !subjective =}-) "we"
     (!Noun = we))

((!plural-pronoun !first !objective =}-) "us"
     (!Noun = us))

(((!singular-pronoun !plural-pronoun ]-)
  !second
  (!subjective !objective ]-)
  =}-) "you"
     (!Noun = you))

((!singular-pronoun !third !subjective !masculine =}-) "he"
     (!Noun = he))

((!singular-pronoun !third !objective !masculine =}-) "him"
     (!Noun = him))

((!singular-pronoun !third !subjective !feminine =}-) "she"
     (!Noun = she))
```

```
((!singular-pronoun
  !third
  (!objective !possessive-determiner ]-)
  !feminine
  =}-) "her"
     (!Noun = her))

((!plural-pronoun !third !subjective =}-) "they"
     (!Noun = they))

((!plural-pronoun !third !objective =}-) "them"
     (!Noun = them))

((!singular-pronoun
  !third
  (!subjective !objective ]-)
  !neuter
  =}-) "it"
     (!Noun = it))

((!singular-pronoun !first !possessive-pronoun =}-) "mine"
     (!Noun = mine))

((!plural-pronoun !first !possessive-pronoun =}-) "ours"
     (!Noun = ours))

(((!plural-pronoun !singular-pronoun ]-)
  !second
  !possessive-pronoun
  =}-) "yours"
     (!Noun = yours))

((!singular-pronoun
  !third
  (!possessive-pronoun !possessive-determiner ]-)
  !masculine
  =}-) "his"
     (!Noun = his))

((!singular-pronoun
  !third
  !possessive-pronoun
  !feminine
  =}-) "hers"
     (!Noun = hers))

((!plural-pronoun !third !possessive-pronoun =}-) "theirs"
     (!Noun = theirs))

((!singular-pronoun !first !possessive-determiner =}-) "my"
     (!Noun = my))

((!plural-pronoun !first !possessive-determiner =}-) "our"
     (!Noun = our))
```

```
(((!singular-pronoun !plural-pronoun ]-)
  !second
  !possessive-determiner
  =}-) "your"
     (!Noun = your))

((!singular-pronoun
  !third
  !neuter
  !possessive-determiner
  =}-) "its"
     (!Noun = its))

((!plural-pronoun
  !third
  !possessive-determiner
  =}-) "their"
     (!Noun = their))

((!animate !subjective =}-) "who"
     (!Noun = who))

((!animate
  (!possessive-pronoun !possessive-determiner ]-)
  =}-) "whose"
     (!Noun = whose))

((!near !singular-pronoun =}-) "this"
     (!Noun = this))

((!near !plural-pronoun =}-) "these"
     (!Noun = these))

((!far !singular-pronoun =}-) "that"
     (!Noun = that))

((!far !plural-pronoun =}-) "those"
     (!Noun = those))

((!sweet !non-possessive =}-) "sweet"
     (!Noun = sweet))

((!gazebo !non-possessive =}-) "gazebo"
     (!Noun = gazebo))

((!cwren !non-possessive =}-) "CWren"
     (!Noun = |Sir Christopher Wren|))

((!cwren !possessive =}-) "CWrens"
     (!Noun = |Sir Christopher Wren's|))

((!daddy !non-possessive =}-) "Daddy"
     (!Noun = Daddy))
```

```
((!taj-mahal !non-possessive =}-) "the-Taj-Mahal"
     (!Noun = |the Taj Mahal|))

((!table !non-possessive =}-) "table"
     (!Noun = table))

((!blockA !non-possessive =}-) "A"
     (!Noun = A))

((!blockB !non-possessive =}-) "B"
     (!Noun = B))

((!blockC !non-possessive =}-) "C"
     (!Noun = C))

((!blockD !non-possessive =}-) "D"
     (!Noun = D))

((!mary !non-possessive =}-) "Mary"
     (!Noun = Mary))

((!mary !possessive =}-) "Marys"
     (!Noun = |Mary's|))

((!upstairs !singular !non-possessive =}-) "upstairs"
     (!Noun = upstairs))

((!headache !singular !non-possessive =}-) "headache"
     (!Noun = headache))

((!headache !plural !non-possessive =}-) "headaches"
     (!Noun = headaches))

((!infection !singular !non-possessive =}-) "infection"
     (!Noun = infection))

((!fever !singular !non-possessive =}-) "fever"
     (!Noun = fever))

((!possibility !singular !non-possessive =}-) "possibility"
     (!Noun = possibility))

((!floor !singular !non-possessive =}-) "floor"
     (!Noun = floor))

((!floor !plural !non-possessive =}-) "floors"
     (!Noun = floors))

((!basement-floor
  !singular
  !non-possessive
  =}-) "basement-floor"
     (!Noun = |basement floor|))
```

```
((!finished-flooring
  !singular
  !non-possessive
  =}-) "finished-flooring"
     (!Noun = |finished flooring|))

((!flooring !singular !non-possessive =}-) "flooring"
     (!Noun = flooring))

((!painter !singular !non-possessive =}-) "painter"
     (!Noun = painter))

((!painting !singular !non-possessive =}-) "painting"
     (!Noun = painting))

((!carpentry !singular !non-possessive =}-) "carpentry"
     (!Noun = carpentry))

((!plasterer !singular !non-possessive =}-) "plasterer"
     (!Noun = plasterer))

((!plaster-board
  !singular
  !non-possessive
  =}-) "plaster-board"
     (!Noun = |plaster board|))

((!plan !singular !non-possessive =}-) "plan"
     (!Noun = plan))

((!task-decorate
  !singular
  !non-possessive
  =}-) "decorate-task"
     (!Noun = |DECORATE TASK|))
```

8. The conjunction network

This network of conjunctive expressions was based on (Halliday and Hasan, 1976, pp. 242-243). The least delicate systems are from the clause network of (Mann/Halliday) which interface well with the table of conjunctive relations in (Halliday and Hasan, 1976). Thus this network takes the form of an extention of the clause network rather than the separate network at the work rank.

```
((clause -{=) CONJUNCTION)

((CONJUNCTION -[) non-conjuncted)
```

```
((CONJUNCTION -[) conjuncted)

((conjuncted -{=) CONJUNCTION-ARGUMENT)

((CONJUNCTION-ARGUMENT -[) thesis)

((CONJUNCTION-ARGUMENT -[) proposition)

((conjuncted -{=) CONJUNCTION-TYPE)

((CONJUNCTION-TYPE -[) temporal-type
     (Conjunct / Time))

((temporal-type thesis textual-time =}-) thesis-temporal
     (Temporal / Time))

((thesis-temporal -[) tt-simple)

((tt-simple -[) thesis-sequential
     (Time = then))

((tt-simple -[) thesis-simultaneous
     (Time = |at the same time|))

((tt-simple -[) thesis-preceding
     (Time = |before that|))

((tt-simple -[) thesis-succeeding
     (Time = |after that|))

((tt-simple -[) thesis-initial
     (Time = first))

((tt-simple -[) thesis-conclusion
     (Time = finally))

((thesis-temporal -[) tt-complex)

((tt-complex -[) thesis-immediate
     (Time = |at once|))

((tt-complex -[) thesis-interrupted
     (Time = soon))

((tt-complex -[) thesis-repetitive
     (Time = |next time|))

((tt-complex -[) thesis-durative
     (Time = meanwhile))

((tt-complex -[) thesis-terminal
     (Time = |until then|))
```

```
((tt-complex -[ ) thesis-punctilliar
     (Time = |at this moment|))

((temporal-type proposition =}-) prop-temporal)

((prop-temporal -[ ) prop-sequential
     (Time = next))

((prop-temporal -[ ) prop-conclusion
     (Time = finally))

((prop-temproal -[ ) here-and-now)

((here-and-now -[ ) prop-past
     (Time = |up to now|))

((here-and-now -[ ) prop-present
     (Time = |at this point|))

((here-and-now -[ ) prop-future
     (Time = |from now on|))

((prop-temporal -[ ) summary)

((summary -[ ) prop-summarizing
     (Time = |to sum up|))

((summary -[ ) prop-resumption
     (Time = |to return to the point|))

((CONJUNCTION-TYPE -[ ) causal
     (Conjunct / Causal))

((causal thesis textual-reason =}-) thesis-causal
     (Causal / Reason))

((causal proposition =}-) prop-causal)

(((thesis-causal prop-causal ]-) -[ ) causal-general)

((causal-general -[ ) gen-simple
     (Causal = so))

((causal-general -[ ) gen-emphatic
     (Causal = consequently))

(((thesis-causal prop-causal ]-) -[ ) causal-specific)

((causal-specific -[ ) causal-reason
     (Causal = |on account of this|))

((causal-specific -[ ) causal-result
     (Causal = |as a result|))
```

```
((causal-specific -[ ) causal-purpose
     (Causal = |with this in mind|))

((causal-specific -[ ) causal-prestatement
     (Causal = since))

((thesis-causal -[ ) thesis-conditional)

((prop-causal -[ ) prop-specific)

((prop-specific -[ ) prop-reason
     (Causal = |on this basis|))

((prop-specific -[ ) prop-result
     (Causal = |arising out of this|))

((prop-specific -[ ) prop-purpose
     (Causal = |to this end|))

((prop-causal -[ ) reversed-causal
     (Causal = because))

((prop-causal -[ ) prop-respective)

((prop-respective -[ ) resp-direct
     (Causal = |in this respect|))

((prop-respective -[ ) resp-rev-polarity
     (Causal = otherwise))

((prop-causal -[ ) prop-conditional)

(((thesis-conditional prop-conditional ]-) -[ ) conditional)

((conditional -[ ) cond-antecedent
     (Causal = if))

((conditional -[ ) cond-simple
     (Causal = then))

((conditional -[ ) cond-emphatic
     (Causal = |in that case|))

((conditional -[ ) cond-generalized
     (Causal = |under the circumstances|))

((conditional -[ ) cond-rev-polarity
     (Causal = |under other circumstances|))

((CONJUNCTION-TYPE -[ ) additive
     (Conjunct / Additive))

((additive -[ ) add-simple)
```

```
((add-simple -[ ] simp-add
     (Additive = and))

((add-simple -[ ] simp-neg
     (Additive = nor))

((add-simple -[ ] simp-alternative
     (Additive = or))

((additive -[ ] add-emphatic)

((add-emphatic -[ ] emph-add
     (Additive = besides))

((add-emphatic -[ ] emph-alternative
     (Additive = alternatively))

(((additive proposition =}-) -[ ] add-de-emphatic
     (Additive = |by the way|))

(((additive proposition =}-) -[ ] add-apposition)

((add-apposition -[ ] expository
     (Additive = |I mean|))

((add-apposition -[ ] examplificatory
     (Additive = |for instance|))

(((additive proposition =}-) -[ ] add-comparison)

((add-comparison -[ ] comp-similar
     (Additive = similarly))

((add-comparison -[ ] comp-disimilar
     (Additive = |on the other hand|))

((CONJUNCTION-TYPE -[ ] adversative
     (Conjunct / Advers))

((adversative -[ ] advers-proper)

((advers-proper -[ ] prop-simple
     (Advers = yet))

((advers-proper -[ ] prop-emph
     (Advers = nevertheless))

(((adversative thesis =}-) -[ ] cont-simple
     (Advers = but))

(((adversative thesis =}-) -[ ] cont-emph
     (Advers = however))
```

```
(((adversative proposition =}-) -[] avowal
     (Advers = |as a matter of fact|))

(((adversative proposition =}-) -[] correction)

((correction -[] of-meaning
     (Advers = rather))

((correction -[] of-wording
     (Advers = |I mean|))

(((adversative proposition =}-) -[] dismissal)

((dismissal -[] dismiss-closed
     (Advers = |in any case|))

((dismissal -[] dismiss-open-ended
     (Advers = |at any rate|))

((CONJUNCTION-TYPE -[] continuative
     (Conjunct / Cont))

((continuative -[] changing
     (Cont = now))

((continuative -[] cont-advers
     (Cont = |of course|))

((continuative -[] responsive/explanative
     (Cont = well))

((continuative -[] cont-resumptive
     (Cont = anyway))

((continuative -[] cont-conclusive
     (Cont = |after all|))
```

9. The modal adjunct network

This network is based on (Halliday, 1985, p. 50).

```
(nil !modal-adjunct
     (# ^ !Adj) (!Adj ^ #))

((!modal-adjunct -[] !no-degree)

((!no-degree -[] !admissive)

((!admissive -[] !frankness
     (!Adj = frankly))
```

```
((!admissive -[] !honesty
     (!Adj = |to be honest|))

((!admissive -[] !truth
     (!Adj = |to tell you the truth|))

((!no-degree -[] !assertive)

((!assertive -[] !lack-of-doubt
     (!Adj = |without any doubt|))

((!assertive -[] !seriousness
     (!Adj = seriously))

((!assertive -[] !reality
     (!Adj = really))

((!assertive -[] !should-believe
     (!Adj = |believe me|))

((!modal-adjunct -[] !degree)

((!degree -{=) TYPE)

((!degree -{=) DEGREE)

((TYPE -[] !likely)

((TYPE -[] !obvious)

((TYPE -[] !often)

((TYPE -[] !typical)

((TYPE -[] !presumable)

((TYPE -[] !desirable)

((TYPE -[] !constant)

((TYPE -[] !valid)

((TYPE -[] !sensible)

((TYPE -[] !expected)

((DEGREE -[] !high)

((DEGREE -[] !medium)

((DEGREE -[] !low)

((!likely !high =}-) "certainly"
     (!Adj = certainly))
```

```
((!likely !medium =}-) "probably"
     (!Adj = probably))

((!likely !low =}-) "possibly"
     (!Adj = possibly))

((!obvious !high =}-) "obviously"
     (!Adj = obviously))

((!obvious !medium =}-) "maybe"
     (!Adj = maybe))

((!obvious !low =}-) "perhaps"
     (!Adj = perhaps))

((!often !high =}-) "always"
     (!Adj = always))

((!often !medium =}-) "usually"
     (!Adj = usually))

((!often !low =}-) "sometimes"
     (!Adj = sometimes))

((!typical !high =}-) "often"
     (!Adj = often))

((!typical !medium =}-) "for-the-most-part"
     (!Adj = |for the most part|))

((!typical !low =}-) "seldom"
     (!Adj = seldom))

((!presumable !high =}-) "no-doubt"
     (!Adj = |no doubt|))

((!presumable !medium =}-) "presumably"
     (!Adj = presumably))

((!presumable !low =}-) "apparently"
     (!Adj = apparently))

((!desirable !high =}-) "to-my-delight"
     (!Adj = |to my delight|))

((!desirable !medium =}-) "fortunately"
     (!Adj = fortunately))

((!desirable !low =}-) "unfortunately"
     (!Adj = unfortunately))

((!constant !high =}-) "initially"
     (!Adj = initially))
```

```
((!constant !medium =}-) "provisionally"
     (!Adj = provisionally))

((!constant !low =}-) "tentatively"
     (!Adj = tentatively))

((!valid !high =}-) "on-the-whole"
     (!Adj = |on the whole|))

((!valid !medium =}-) "in-general-terms"
     (!Adj = |in general terms|))

((!valid !low =}-) "strictly-speaking"
     (!Adj = |strictly speaking|))

((!sensible !high =}-) "wisely"
     (!Adj = wisely))

((!sensible !medium =}-) "understandably"
     (!Adj = understandably))

((!sensible !low =}-) "by-mistake"
     (!Adj = |by mistake|))

((!expected !high =}-) "as-expected"
     (!Adj = |as expected|))

((!expected !medium =}-) "by-chance"
     (!Adj = |by chance|))

((!expected !low =}-) "amazingly"
     (!Adj = amazingly))
```

10. The adjective network

```
(nil !adjective
     (# ^ !Adj) (!Adj ^ #))

((!adjective -[ ) !clear
     (!Adj = clear))

((!adjective -[ ) !red
     (!Adj = red))

((!adjective -[ ) !big
     (!Adj = big))

((!adjective -[ ) !small
     (!Adj = small))

((!adjective -[ ) !severe
     (!Adj = severe))
```

```
((!adjective -[ ) !mild
     (!Adj = mild))

((!adjective -[ ) !tangy
     (!Adj = tangy))
```

11. The adverb network

```
(nil !adverb
     (# ^ !Adv) (!Adv ^ #))

((!adverb -[ ) !quickly
     (!Adv = quickly))

((!adverb -[ ) !immediately
     (!Adv = immediately))

((!adverb -[ ) !now
     (!Adv = now))
```

12. The preposition network

This fragment of a preposition network was taken from (Quirk and Greenbaum, 1973, p. 146).

```
(nil !preposition
     (# ^ !Prep) (!Prep ^ #))

((!preposition -{=) POS/NEG)

((POS/NEG -[ ) !prep-positive)

((POS/NEG -[ ) !prep-negative)

((!preposition -{=) DIR/POS)

((DIR/POS -[ ) !prep-direction)

((DIR/POS -[ ) !prep-position)

((!preposition -{=) DIMENSION)

((DIMENSION -[ ) !point)

((DIMENSION -[ ) !line/surface)

((DIMENSION -[ ) !area/volume)

((!prep-positive !prep-direction !point =}-) "to"
     (!Prep = to))
```

```
((!prep-positive !prep-position !point =}-) "at"
     (!Prep = at))

((!prep-negative !prep-direction !point =}-) "from"
     (!Prep = from))

((!prep-negative !prep-position !point =}-) "away-from"
     (!Prep = |away from|))

((!prep-positive !prep-direction !line/surface =}-) "onto"
     (!Prep = onto))

((!prep-positive !prep-position !line/surface =}-) "on"
     (!Prep = on))

((!prep-negative !line/surface =}-) "off"
     (!Prep = off))

((!prep-positive !prep-direction !area/volume =}-) "into"
     (!Prep = into))

((!prep-positive !prep-position !area/volume =}-) "in"
     (!Prep = in))

((!prep-negative !area/volume =}-) "out-of"
     (!Prep = |out of|))
```

13. The semantic stratum

The semantic system network presented here contains the house building semantics (see Appendix B Section 2) and the threat semantics (see Section 3.6 and Appendix B Section 3).

```
(nil $emantics)

;;; The house building semantics

(($emantics -[) $build
     (# ^ $Build)
     ($Build ^ #)
     ($Build : non-benefactive)
     ($Build : unmarked-positive)
     ($Build : non-attitudinal)
     ($Build : non-present-in)
     ($Build : textual-theme)
     ($Build : non-place)
     ($Build<Goal : non-possessive-nom)
     ($Build<Goal : determined)
     ($Build<Goal : non-quantified)
     ($Build<Goal : non-selective)
     ($Build<Goal : noun))
```

```
((($build -{=) ENABLEMENT)

((ENABLEMENT -[) $non-enabled
     ($Build : present))

(($non-enabled -[) $untied
     ($Build : non-past-in))

(($untied -[) $initial-action
     ($Build : thesis-initial))

(($untied -[) $simultaneous-action
     ($Build : thesis-simultaneous))

(($non-enabled -[) $enabling
     ($Build : past-in) ($Build : non-time))

(($enabling -[) $first-enabling
     ($Build : thesis-conditional) ($Build : cond-antecedent))

(($enabling -[) $another-enabling
     ($Build : simp-add))

((ENABLEMENT -[) $enabled
     ($Build : non-past-in)
     ($Build : modal)
     ($Build<Modal : !can)
     ($Build : non-time)
     ($Build : unmarked-declarative-theme))

(($enabled -[) $single-enabler
     ($Build : thesis-succeeding))

(($enabled -[) $multiple-enabler
     ($Build : thesis-conditional) ($Build : cond-simple))

(($build -{=) DISCOURSE-TYPE)

((DISCOURSE-TYPE -[) $builder-oriented
     ($Build : residual) ($Build : operative))

((DISCOURSE-TYPE -[) $action-oriented)

(($build -{=) ADDRESSEE)

((ADDRESSEE -[) $not-addressee-builder
     ($Build : unmarked-declarative-theme)
     ($Build : nominal-subject))

((ADDRESSEE -[) $addressee-builder
     ($Build : residual) ($Build : operative))

(($build -{=) COMPONENT-NUMBER)
```

```
((COMPONENT-NUMBER -[) $singular-component
     ($Build<Goal : singular))

((COMPONENT-NUMBER -[) $plural-component
     ($Build<Goal : plural))

((COMPONENT-NUMBER -[) $mass-component
     ($Build<Goal : mass))

(($build -{=) BUILDER-NUMBER)

((BUILDER-NUMBER -[) $one-builder)

((BUILDER-NUMBER -[) $several-builders)

(($build -{=) ACTION)

((ACTION -[) $painting
     ($Build : dispositive)
     ($Build<Process : !-do-)
     ($Build<Goal<Head : !painting))

((ACTION -[) $create-basement-floor
     ($Build : creative)
     ($Build<Process : !pour)
     ($Build<Goal<Head : !basement-floor))

((ACTION -[) $put-up-plaster-board
     ($Build : dispositive)
     ($Build<Process : !fasten)
     ($Build<Goal<Head : !plaster-board))

((ACTION -[) $do-finished-flooring
     ($Build : dispositive)
     ($Build<Process : !lay)
     ($Build<Goal<Head : !finished-flooring))

((ACTION -[) $carpentry
     ($Build : dispositive)
     ($Build<Process : !finish)
     ($Build<Goal<Head : !carpentry))

((ACTION -[) $sanding-floor
     ($Build : dispositive)
     ($Build<Process : !sand)
     ($Build<Goal<Head : !floor))

((ACTION -[) $varnishing-floor
     ($Build : dispositive)
     ($Build<Process : !varnish)
     ($Build<Goal<Head : !floor))
```

```
((($addressee-builder $untied -<>) $addressee-command
     ($Build : unmarked-imperative-theme)
     ($Build : imperative-subject-explicit))

(($addressee-builder ($enabled $enabling ]-) -<>) $addressee-check
     ($Build : unmarked-declarative-theme)
     ($Build : addressee-subject))

((($builder-oriented $one-builder =}-)
  ($action-oriented $singular-component =}-)
  ]-) $single-subject
     ($Build : singular-subject))

((($builder-oriented $several-builders =}-)
  ($action-oriented $plural-component =}-)
  ]-) $plural-subject
     ($Build : plural-subject))

(($not-addressee-builder
  $action-oriented
  $mass-component
  =}-) $mass-subject
     ($Build : mass-subject))

(($not-addressee-builder $action-oriented -<>) $builder-unmentioned
     ($Build : non-agentive) ($Build : non-residual))

(($not-addressee-builder $builder-oriented -<>) $builder-mentioned
     ($Build<Actor : non-possessive-nom)
     ($Build<Actor : non-quantified)
     ($Build<Actor : non-selective)
     ($Build<Actor : determined)
     ($Build<Actor : noun))

(($builder-mentioned $painting =}-) $painter
     ($Build<Actor<Head : !painter))

(($builder-mentioned
  ($do-finished-flooring $carpentry ]-)
  =}-) $carpenter
     ($Build<Actor<Head : !carpenter))

(($builder-mentioned $put-up-plaster-board =}-) $plasterer
     ($Build<Actor<Head : !plasterer))

;;; The threat semantics from (Halliday 1978, pp. 82-84).

(($emantics -[) $threat
     ($Threat : non-past-in)
     ($Threat : dispositive)
     ($Threat ^ #))

(($threat -[) $loss-of-privilege
     (# ^ $Threat) ($Threat : non-textual-theme))
```

```
(($loss-of-privilege -[ ) $command
     ($Threat : middle)
     ($Threat : jussive)
     ($Threat : non-benefactive)
     ($Threat : unmarked-imperative-theme)
     ($Threat : proper-subject)
     ($Threat : non-residual)
     ($Threat : non-present-in)
     ($Threat : operative)
     ($Threat : unmarked-positive)
     ($Threat : place)
     ($Threat<Process : !go))

(($command -[ ) $get-attention
     ($Threat : imperative-subject-explicit))

(($command -[ ) $unmarked-command
     ($Threat : imperative-subject-implicit))

(($loss-of-privilege -[ ) $decision
     ($Threat : unmarked-declarative-theme)
     ($Threat : non-attitudinal))

(($decision -[ ) $rejection
     ($Threat : unmarked-positive)
     ($Threat : place)
     ($Threat : non-benefactive))

(($decision -[ ) $deprivation
     ($Threat : marked-negative)
     ($Threat : non-place)
     ($Threat : residual)
     ($Threat<Process : !give)
     ($Threat<Medium : non-possessive-nom)
     ($Threat<Medium : singular)
     ($Threat<Medium : noun)
     ($Threat<Medium : non-quantified)
     ($Threat<Medium : determined)
     ($Threat<Medium : non-add)
     ($Threat<Medium<Head : !sweet))

(($decision -[ ) $resolution
     ($Threat : present-in))

(($decision -[ ) $obligation
     ($Threat : passive-modulation) ($Threat : necessary))

(($decision -[ ) $mother-centred-decision
     ($Threat : speaker-subject) ($Threat : operative))

(($decision -[ ) $child-centred-decision
     ($Threat : addressee-subject))
```

```
(($resolution -[] $pending
     ($Threat : present))

((($pending $command ]-) -[] $unmarked-time
     ($Threat : non-time))

((($pending $command ]-) -[] $immediate
     ($Threat : non-textual-time)
     ($Threat<Temporal : !now))

(($resolution -[] $deferred
     ($Threat : future) ($Threat : non-time))

(($emantics -[] $at-home)

(($threat -[] $punishment
     ($Threat : non-benefactive)
     ($Threat : unmarked-declarative-theme)
     ($Threat : non-place)
     ($Threat : non-attitudinal)
     ($Threat : non-present-in)
     ($Threat : residual)
     ($Threat : unmarked-positive)
     ($Threat : future))

(($punishment -[] $adult-centred-punishment
     ($Threat : operative)
     ($Threat<Goal : non-possessive-nom)
     ($Threat<Goal : personal)
     ($Threat<Goal : singular)
     ($Threat<Goal<Head : !second)
     ($Threat<Goal<Head : !objective))

(($punishment -[] $child-centred-punishment
     ($Threat : addressee-subject))

(($punishment -[] $chastisement)

(($punishment -[] $authority-figure
     ($Threat<Actor : non-possessive-nom)
     ($Threat<Actor : noun))

(($punishment -[] $unconditional
     (# ^ $Threat) ($Threat : non-textual-theme))
```

```
((Spunishment -[ ) Sconditional
     (# ^ $Cond)
     ($Cond ^ $Threat)
     ($Cond : non-benefactive)
     ($Cond : non-place)
     ($Cond : non-time)
     ($Cond : addressee-subject)
     ($Cond : operative)
     ($Cond : residual)
     ($Cond : non-past-in)
     ($Cond : non-present-in)
     ($Cond : dispositive)
     ($Cond<Process : !-do-)
     ($Cond<Goal : non-possessive-nom)
     ($Cond<Goal : singular)
     ($Cond<Goal : determiner-head)
     ($Cond<Goal : far))

((Sconditional -[ ) $explicit-repetition)

((Sconditional -[ ) $non-repetitive
     ($Threat : non-time))

((Sconditional -[ ) $logical-cond
     ($Cond : textual-theme)
     ($Threat : non-textual-theme)
     ($Cond : thesis-conditional)
     ($Cond : cond-antecedent)
     ($Cond : unmarked-positive)
     ($Cond : present))

((Sconditional -[ ) $non-logical-cond
     ($Threat : textual-theme)
     ($Cond : non-textual-theme)
     ($Cond : unmarked-negative))

(($non-logical-cond -[ ) $threatening-reason
     ($Threat : reversed-causal))

(($non-logical-cond -[ ) $threatening-alternative
     ($Threat : simp-alternative))

(($non-logical-cond -[ ) $straight-threat)

(($non-logical-cond -[ ) $exclamatory-cond
     ($Cond : unmarked-imperative-theme)
     ($Cond : proper-subject)
     ($Cond : imperative-subject-implicit))

(($non-logical-cond -[ ) $explanatory-cond
     ($Cond : modal)
     ($Cond : addressee-subject)
     ($Cond<Modal : !must))
```

```
((Sdeprivation
  Smother-centred-decision
  -<>) Smother-deprivation
     (SThreat : ben-med)
     (SThreat<Beneficiary : non-possessive-nom)
     (SThreat<Beneficiary : personal)
     (SThreat<Beneficiary : singular)
     (SThreat<Beneficiary<Head : !second)
     (SThreat<Beneficiary<Head : !objective))

((Sdeprivation
  Schild-centred-decision
  -<>) Schild-deprivation
     (SThreat : non-agentive) (SThreat : benereceptive))

((Srejection
  Smother-centred-decision
  -<>) Smother-rejection
     (SThreat : residual)
     (SThreat<Process : !take)
     (SThreat<Goal : non-possessive-nom)
     (SThreat<Goal : personal)
     (SThreat<Goal : singular)
     (SThreat<Goal<Head : !second)
     (SThreat<Goal<Head : !objective))

((Srejection Schild-centred-decision -<>) Schild-rejection
     (SThreat : non-residual)
     (SThreat<Process : !go)
     (SThreat : non-ranged))

((Sexplanatory-cond Slogical-cond ]-) Sstated-cond
     (SCond : unmarked-declarative-theme)
     (SCond : non-attitudinal))

((Sstraight-threat
  Sexplicit-repetition
  =}-) Srepeat-straight
     (SThreat : thesis-repetitive))

((Sadult-centred-punishment
  Schastisement
  -<>) Smother-punishes
     (SThreat : speaker-subject))

((Sadult-centred-punishment
  Sauthority-figure
  -<>) Sauthority-punishes
     (SThreat : nominal-subject))

((Schild-centred-punishment
  Sauthority-figure
  -<>) Spunished-by-authority
     (SThreat : agentive))
```

```
((($child-centred-punishment
  $chastisement
  -<>) $punished-by-mother
     ($Threat : non-agentive))

(($punishment -[) $smack
     ($Threat<Process : !smack))

(($authority-figure -[) $daddy
     ($Threat<Actor<Head : !daddy)
     ($Threat<Actor : singular)
     ($Threat<Actor : non-determined)
     ($Threat<Actor : non-quantified))

((($rejection $command ]-) $at-home =}-) $threat-at-home
     ($Threat<Spatial<Range : noun)
     ($Threat<Spatial<Range : singular)
     ($Threat<Spatial<Range : non-determined)
     ($Threat<Spatial<Range : non-quantified)
     ($Threat<Spatial<Range : non-possessive-nom)
     ($Threat<Spatial<Range<Head : !upstairs)
     ($Threat<Spatial : merged))
```

Appendix D: Program Listing

1. The initialization file

```
(princ '|... initialize SLANG ...|)
(terpri)
(setsyntax '|#| 'vcharacter)
(setsyntax '|"| 'vcharacter)

(watch 0) ; no trace of productions firing

(defun reload (fn) ; load and run a new example
  (remove *) ; clear working memory
  (load fn)  ; a file containing goals
  (princ '|================|) (terpri)
  (princ '|     SLANG:     |) (terpri)
  (princ '|================|) (terpri)
  (make sub-judice 0) ; initialize root node
  (run))


(external PRESELECT GOAL ASSERT)

(vector-attribute for)

(strategy mea) ; use the means-end analysis strategy


;;; The literalize statements define the attributes
;;; for the various elements.


(literalize adjacent to is mom)

(literalize conflate fun with mom)

(literalize expand fun into mom)

(literalize lexify fun as mom)

(literalize preselect feature for mom)

(literalize hub of is super mom lex output?)
```

2. Realization productions

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This file contains rules relating to the realization in  ;
; systemic grammar. They build and manipulate the syntactic;
; structures associated with a text.                       ;
;                                                          ;
; The primary data structure is the "hub" which is a node  ;
; in the syntactic structure tree.                         ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;*************************************************************
; Realization rules take grammatical functions as           ;
; arguments and these must be associated with hubs.         ;
; The following rules create hubs for functions that        ;
; do not yet have one.                                      ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;IF   there is a conflate lexify or expand realization
;;;     rule whose first argument does not have a hub,
;;;THEN create a unique hub for it.
(p new-hub::first-arg
  (<< conflate lexify expand >> ^fun <f> ^mom <m>)
 -(hub ^of <f> ^mom <m>)
-->
  (make hub ^of <f> ^is (genatom) ^mom <m>))


;;;IF   there is a conflate realization rule whose
;;;     second argument does not have a hub,
;;;THEN create a unique hub for it.
(p new-hub::second-arg:conflate
  (conflate ^with <f> ^mom <m>)
 -(hub ^of <f> ^mom <m>)
-->
  (make hub ^of <f> ^is (genatom) ^mom <m>))


;;;IF   there is an expand realization rule whose
;;;     second argument does not have a hub,
;;;THEN create a unique hub for it.
(p new-hub::second-arg:expand
  (expand ^into <f> ^mom <m>)
 -(hub ^of <f> ^mom <m>)
-->
  (make hub ^of <f> ^is (genatom) ^mom <m>))
```

```
;*************************************************************
; The following rules actually implement the realization.   ;
; The above rules have guaranteed that the functions        ;
; involved are associated with hubs                         ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;IF   an expand rule is encountered, expanding a function
;;;     into subfunctions
;;;THEN put the hub of the expanded function as the ^super
;;;     of the hub of the subfunction.
(p expand
 {(expand ^fun <f> ^into <subf> ^mom <m>)       <expand>}
  (hub ^of <f> ^is <h> ^mom <m>)
 {(hub ^of <subf> ^mom <m>)                  <hub-of-subf>}
-->
  (modify <hub-of-subf> ^super <h>)
  (remove <expand>))


;;;IF   a conflate rule is encountered and the functions
;;;     have different hubs,
;;;THEN substitute the hub of the first for all
;;;     instances of the hub of the second by setting
;;;     the task change-hub.
(p conflate
 {(conflate ^fun <f1> ^with <f2> ^mom <m>)      <conflate>}
  (hub ^of <f1> ^is <h1> ^mom <m>)
  (hub ^of <f2> ^is { <h2> <> <h1> } ^mom <m>)
-->
  (remove <conflate>)
  (make task change-hub <h2> to <h1>))


;;;IF   a preselection rule is encountered
;;;THEN pass the whole thing to a lisp operator that
;;;     handles this.
;;; [This is necessary because the number of arguments
;;;  is arbitrary.]
(p preselect
 {(preselect ^feature <feature> ^mom <m>)       <preselect>}
-->
  (call PRESELECT (substr <preselect> 1 inf))
  (remove <preselect>))


;;;IF   a lexify rule is encountered,
;;;THEN associate the lexical item with the function's hub.
(p lexify
 {(lexify ^fun <f> ^as <lex> ^mom <m>)          <lexify>}
 {(hub ^of <f> ^is <h> ^mom <m> ^output? <> yes) <hub>}
-->
  (modify <hub> ^lex <lex> ^output? no)
  (remove <lexify>))
```

```
;*************************************************************
; The following rules are used to conflate to functions     ;
; (see above). They substitute one function for another     ;
; throughout working memory.                                ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;IF   any functions are associated with the old hub,
;;;THEN associate them with the new one instead.
(p change-hub::functions
  (task change-hub <old> to <new>)
 {(hub ^is <old>)                                  <hub>}
-->
  (modify <hub> ^is <new>))


;;;IF   any value attribute statements have the old hub
;;;     in the ^mom field,
;;;THEN change it to the new one.
(p change-hub::mothers:value-attribute
  (task change-hub <old> to <new>)
 {(<< hub adjacent >> ^mom <old>)          <hub/adjacent>}
-->
  (modify <hub/adjacent> ^mom <new>))


;;;IF   any vector statements have the old hub
;;;     in the last (mom) field,
;;;THEN change it to the new one.
(p change-hub::mothers:vector
  (task change-hub <old> to <new>)
 {(<< chosen goal >> {} <old>)                     <vector>}
-->
  (modify <vector> ^3 <new>))


;;;IF   any ^super fields have the old hub
;;;THEN change it to the new one.
(p change-hub::super
  (task change-hub <old> to <new>)
 {(hub ^super <old>)                               <hub>}
-->
  (modify <hub> ^super <new>))
```

3. The support system

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; this file contains the support system --                              ;;;
;;; productions that don't really have any intimate                       ;;;
;;; connection to systemic grammar but are necessary for                  ;;;
;;; the system to work.                                                   ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;IF   a task is no longer appropriate
;;;     (it matches no other rules),
;;;THEN delete it.
(p remove-task
 {(task)                                                  <task>}
-->
  (remove <task>))


;;;IF   a node is under-consideration (sub-judice),
;;;THEN make sure its supernode is also under consideration.
(p fill-scope::super
  (sub-judice <sj>)
  (hub ^is <sj> ^super { <super> <> nil })
 -(sub-judice <super>)
-->
  (make sub-judice <super>))


;;;IF   a hub has been output,
;;;THEN make sure all the wm elements for that hub are
;;;     marked accordingly.
(p spread-output
  (hub ^is <h> ^output? yes)
 {(hub ^is <h> ^output? <> yes)          <out-of-date>}
-->
  (modify <out-of-date> ^output? yes))


;;;IF   a lexical item has been associated with a hub,
;;; and that hub is sub-judice,
;;;THEN output that lexical item
;;; and mark the hub as output.
(p output
 {(hub ^is <h> ^lex { <l> <> nil } ^output? no)        <hub>}
  (sub-judice <h>)
-->
  (write <l> (crlf))
  (modify <hub> ^output? yes))
```

```
;************************************************************
; The following rules move around the boundaries of what    ;
; is under consideration (sub-judice). Hubs (structure      ;
; nodes) are marked for consideration in a depth-first      ;
; left to right fashion.                                    ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;IF   a node is sub-judice and not output,
;;;THEN declare its leftmost child (if there is one)
;;;     sub-judice.
(p move::down:#
  (sub-judice <sj>)
  (adjacent ^to # ^is <f> ^mom <sj>)
  (hub ^of <f> ^is <h> ^output? <> yes ^mom <sj>)
 -(hub ^is <sj> ^output? yes)
-->
  (make sub-judice <h>))


;;;IF   a node is sub-judice and not output,
;;;THEN declare its leftmost subnode (if there is one)
;;;     sub-judice.
(p move::down:%
  (sub-judice <sj>)
  (adjacent ^to % ^is <f> ^mom <m>)
  (hub ^of <f> ^is <h> ^super <sj> ^output? <> yes ^mom <m>)
 -(hub ^is <sj> ^output? yes)
-->
  (make sub-judice <h>))


;;;IF   a hub has just been output
;;;THEN declare the node adjacent to it (if there is one)
;;;     sub-judice.
(p move::across
  (hub ^of <f> ^output? yes ^mom <m>)
  (adjacent ^to <f> ^is <f1> ^mom <m>)
  (hub ^of <f1> ^is <h1> ^output? <> yes ^mom <m>)
-->
  (make sub-judice <h1>))


;;;IF   the rightmost child of a node has just been output,
;;;THEN declare the node output
(p move::up:#
  (adjacent ^to <f> ^is # ^mom <m>)
  (hub ^of <f> ^output? yes ^mom <m>)
 {(hub ^is <m> ^output? <> yes)                    <mother>}
-->
  (modify <mother> ^output? yes))
```

```
;;;IF   the rightmost subnode has just been output,
;;;THEN declare the supernode output.
(p move::up:%
  (adjacent ^to <f> ^is % ^mom <m>)
  (hub ^of <f> ^output? yes ^super <super> ^mom <m>)
 {(hub ^is <super> ^output? <> yes)              <supernode>}
-->
  (modify <supernode> ^output? yes))
```

4. The external LISP operators

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; The following are Franz LISP operators called from within;
; OPS5 for the SLANG project. The functions prefixed with  ;
; '$'are special operators provided by OPS5 and are        ;
; described in the OPS5 manual.                            ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; The function GOAL is invoked by the "call GOAL -- <mom>"
;;; rhs statements. This is meant to be like "make goal --"
;;; except it will not enter goals which are already
;;; present. This keeps working memory clean and pays for
;;; itself by cutting down on redundant work.

(defun GOAL ()
    (or (scanwm ($parameter 1)  ; a feature,
                ($parameter 2)) ; its mom.
        ; if scan unsuccessfully
        ; for a feature then:
      (prog (feature mom)
            (setq feature ($parameter 1))
            (setq mom ($parameter 2))
            ($reset)          ; clear the result element
            ($value 'goal)
            ($value feature) ; put (goal -- <mom>) in
            ($value mom)     ;  the result element.
            ($assert))))     ; put the result element in wm.
```

```
;;; ASSERT is called by "(call ASSERT chosen -- <mom>)"
;;; and is equivalent to "(make chosen --)" except that
;;; it avoids duplicates as above.

(defun ASSERT ()
       ; $parameter 1 is the dummy "chosen"
  (or (scanwm ($parameter 2)    ; a feature,
              ($parameter 3))   ; its mom.
       ; if scan unsuccessfully
       ; for a feature then:
      (prog (feature mom)
            (setq feature ($parameter 2))
            (setq mom ($parameter 3))
            ($reset)            ; clear the result element
            ($value 'chosen)
            ($value feature) ; put (chosen -- <mom>)
            ($value mom)     ; in the result element.
            ($assert))))     ; put the result element in wm.


;;; scanwm checks to see if a feature description is
;;; already in working memory.
;;; scanwm is efficient because only the wm goal elements
;;; are looped through. (get 'chosen 'wmpart*) returns a
;;; list of descriptions of wm elements whose car is
;;; "chosen". This is using the OPS5 hashing mechanism
;;; directly -- but is essential for efficiency reasons.

(defun scanwm (feature mom)
    (do ((features (get 'chosen 'wmpart*) (cdr features))
         (target (list feature mom)))
        ((null features) nil)   ; if not found return nil
        (cond ((equal target (cdaar features))
               (return t)))))
```

```
;;; PRESELECT is called from the preselect rule since the
;;; number of arguments is arbitrarily large.
;;; The argument is a preselection statement containing
;;; are a list of functions that form a path
;;; to a place in structure, and a feature. PRESELECT
;;; descends the structure along this path (creating
;;; structure representing the path if necessary) until the
;;; end of the path. It then sets the feature as a goal.

(defun PRESELECT ()
  (do ((feature ($parameter ($litbind 'feature)))
       (mom ($parameter ($litbind 'mom)))
       (path ; initialize path as the vector ^for field
        (do ((i ($litbind 'for) (add1 i))
             (pathlist nil (append pathlist
                                   (list ($parameter i)))))
            ((> i ($parametercount)) pathlist))
        (cdr path))
       (outgoing-hub
        ($parameter ($litbind 'mom))
        (get-hub (car path) outgoing-hub)))
      ((null path) ; loop until path is finished
       (cond ((not (scanwm feature outgoing-hub))
              ($reset)
              ($value 'goal)
              ($value feature)
              ($value outgoing-hub)
              ($assert))))))


;;; returns a hub-number for function, has the side effect
;;; of creating one if one does not already exist.

(defun get-hub (function mom)
  (do ((hubs (get 'hub 'wmpart*) (cdr hubs))
       (newhub))
      ((null hubs) ; if no such hub, create one
       ($reset)
       ($value 'hub)
       ($tab 'of) ($value function)
       ($tab 'is) (setq newhub (gensym 'h)) ($value newhub)
       ($tab 'mom)($value mom)
       ($assert)
       (return newhub))
      (and (eq (get-att-value (car hubs) 'of) function)
           (eq (get-att-value (car hubs) 'mom) mom)
           (return (get-att-value (car hubs) 'is)))))
```

```
;;; get-att-value returns the value corresponding to an
;;; attribute for a particular element.

(defun get-att-value (elem att)
  (nthelem ($litbind att) (car elem)))
```

5. SNORT

```
;;;  System Network - Ops5 Rule Translator
;;;  ====================================
;;; this file contains operators for translating a
;;; system-network into a set of production rules.

(declare                              ;for compilation
  (setsyntax '|#| 'vcharacter)
  (setsyntax '|[| 'vcharacter)  ;these are needed for
  (setsyntax '|]| 'vcharacter)  ; for systems networks.
  (setsyntax '|"| 'vcharacter)
  (special IN-A-SYSTEM p grammar s))


(setsyntax '|#| 'vcharacter)
(setsyntax '|[| 'vcharacter)  ;these are needed for
(setsyntax '|]| 'vcharacter)  ; systems networks.
(setsyntax '|"| 'vcharacter)


;;; snort is the top level operator. It's args are a
;;; system net and an output file. It simply loops
;;; through the features in the grammar, translating
;;; each one in turn.

(defun snort (grammar file)
  (do ((IN-A-SYSTEM nil) ; a global list of all system
                         ; features.
       (undone (cdr grammar) (cdr undone))
       (next (car grammar) (car undone))
       (p (fileopen (concat file '|.ops|) 'w)))
      ((null next) (close p)) ; when done close file
      (cond ((feature-namep (cadr next))
        ; only translate features not system labels
             (f-p (cadr next)
                  (in-a-systemp next) ; boolean flag
                  (get-ecs next)      ; entry conditions
                  (cddr next))        ; realization rules
))))
```

```
;;; in-a-systemp is a predicate which returns t if a
;;; feature is part of a system, and nil if it is part of
;;; a gate. Its argument is the entire feature description,
;;; entry conditions, name and realization rules.
;;; A feature is "in a system" if it is marked as such (-[),
;;; or if it is a degenerate system ie. it is marked
;;; otherwise ('--' etc.) but is indirectly or directly an
;;; entry condition for a system (my definitions).

(defun in-a-systemp (x)
  (cond ((or (system-namep (cadr x))
             (eq (cadar x) '-[)
             (system-namep (caar x))
             (memq (cadr x) IN-A-SYSTEM)
             (null (car x)) ; the root of the network should
                            ; be treated as if it were
                            ; part of a system.
             (apply 'or (mapcar 'in-a-systemp
                                (right-of (cadr x)
                                          grammar))))
         (setq IN-A-SYSTEM (cons (cadr x) IN-A-SYSTEM))
         t)
        (t nil)))


;;; right-of returns the elements of a grammar to the
;;; immediate right of a feature in a system network.
;;; I.e. those features and systems which have this
;;; feature as an entry condition.

(defun right-of (feature gram)
  (cond ((null gram) nil)
        ((memq feature (just-names (caar gram)))
         (cons (car gram) (right-of feature (cdr gram))))
        (t (right-of feature (cdr gram)))))

;;; get-ecs returns the FEATURE entry conditions of
;;; a feature. If the immediate ec of the feature is a
;;; system label, the ecs of the system are returned.

(defun get-ecs (feature)
  (cond ((eq (cadar feature) '-[)
         (cond ((system-namep (caar feature))
                (ecs-of (caar feature)))
               ((feature-namep (caar feature))
                (car feature))
               (t (caar feature)))) ; an expression
        (t (car feature))))
```

```
;;; ecs-of simply scans through the grammar for a feature
;;; whose name is the argument.
;;; When it finds it, the entry conditions are returned.

(defun ecs-of (name)
  (do ((next (car grammar) (car undone))
       (undone (cdr grammar) (cdr undone)))
      ((eq (cadr next) name) (car next))
      (and (null next) (error name 'not 'found))))


;;; system names are assumed to be all upper case.

(defun system-namep (name)
  (and (symbolp name)
       (< (getcharn name 1) 91)
       (> (getcharn name 1) 64)))


;;; feature names are assumed to be all lower case

(defun feature-namep (name)
  (and (symbolp name)
       (or (memq (getchar name 1) '(! $ - "))
           (and (< (getcharn name 1) 123)
                (> (getcharn name 1) 96)))))
```

```
;;; f-p, feature to production, translates the features from
;;; the system network into OPS5 production-rules.
;;; The hard part is translating the entry conditions.
;;; For this purpose f-p calls decoding operators which
;;; return a "simple form"--a list of conjucts where any
;;; sublists are lists of disjuncts (only one level of
;;; nesting [see decode below]).

(defun f-p (name in-system ecs rrs)
  (prpr (append
          (list 'p)
          (list name)
          ; if this feature is part of a system ...
          ;  ie. if feature will be entered from right
          (cond (in-system
                 (append (list '(sub-judice <mom>))
                         (list (list 'goal name '<mom>))
                         (list '-->)
                        '((modify 2 ^1 chosen))
                         (do ((subgoals (decode ecs
                                                'system
                                                t
                                                name)
                                        (cdr subgoals))
                              (goodies nil))
                             ; when done return goodies
                             ((null subgoals) goodies)
                             (cond ((atom (car subgoals))
                                    ;if not a disjunct
                                    (setq goodies
                                          (cons (list 'call
                                                      'GOAL
                                                (car subgoals)
                                                      '<mom>)
                                                goodies)))))))))
```

```
        ; if entry condition is not a system ...
        ;  ie. if this is a gate
        (t (append
             (mapcar '(lambda (ec)
                        (cond ((atom ec)
                               (list 'chosen
                                     ec
                                     '<mom>))
                              (t ; disjunction
                               (append
                                (list 'chosen
                                      '<<)
                                      ec
                                      (list '>>)
                                      (list '<mom>)
                        ))))
                      (decode ecs 'gate t name))
             (list '-->)
             (list (list 'call
                         'ASSERT
                         'chosen
                         name
                         '<mom>)))))
; in either case, the realization rules are
; done the same ...
(mapcar 'unformat rrs))))
```

```
;;; decode is used to transform the input notation for entry
;;; conditions (including =}- -- -<> etc.) into a 'simple
;;; form' that the higher level operators can use easily.
;;; This simple form is a list of conjucts where any
;;; sublists are disjuncts.
;;; Deeper nesting than this (eg. conjuncts within
;;; disjuncts) cannot be represented in the LHS of a single
;;; OPS5 rule.
;;; The two types of ecs (system and gate) identified by the
;;; 'type' arg are treated differently. It is common for
;;; gates to have very complex ecs, so the OPS5 limitation
;;; must be overcome through the use of intermediate rules.
;;; So decode may pass back "(f1 f2 s0003 f3)" where s0003
;;; is the name of a rule which decode itself (via f-p)
;;; has built embodying the next level of conjunction.
;;; This is done recursively so arbitrarily complex
;;; ecs for gates can be decoded.
;;; Disjunctive entry conditions to systems are
;;; simply ignored (as explained in Section 7.2.2.).

(defun decode (ec type top name)
  (cond ((atom ec) ec)
        ((eq (car (last ec)) ']-)
         (mapcar 'flatten       ; remove nested disjunction
;;; if at top level and it is a disjunct,
;;; just convert it to a one element conjunct and restart.
           (cond (top (decode (list ec '--) type t name))
                 ((not top) ; if not top level ...
                  (mapcar '(lambda (e)
                                  (decode e type nil name))
                          (all-but-last ec))))))
        ((or (memq (car (last ec)) '(=}- -- -<> -{=))
             (and (eq type 'system)
                  (eq (car (last ec)) '-[)))
         (cond (top (mapcar '(lambda (e)
                                  (decode e type nil name))
                            (all-but-last ec)))
               ((and (not top) (eq type 'gate))
                (f-p (setq s (concat name
                                     (intern (gensym 'g))))
                     nil
                     ec
                     nil)
                s)))
        (t (error 'bad 'ec 'to 'decode: ec))))
```

```
;;; unformat converts systemic realization rules
;;; into OPS5 notation.

(defun unformat (rr)
  (cond ((not (atom (cadr rr)))
         (list 'make
               'expand
               '^fun (car rr)
               '^into (caadr rr)
               '^mom '<mom>))
        ((eq (cadr rr) '/) (list 'make
                                 'conflate
                                 '^fun (car rr)
                                 '^with (caddr rr)
                                 '^mom '<mom>))
        ((eq (cadr rr) '=) (list 'make
                                 'lexify
                                 '^fun (car rr)
                                 '^as (caddr rr)
                                 '^mom '<mom>))
        ((eq (cadr rr) '^) (list 'make
                                 'adjacent
                                 '^to (car rr)
                                 '^is (caddr rr)
                                 '^mom '<mom>))
        ((eq (cadr rr) ':)
         (preselection (caddr rr) (car rr)))
        (t (error 'bad 'rr 'format: rr))))


(defun preselection (feature function)
  (do ((next (car (explode function)) (car undone))
       (undone (cdr (explode function)) (cdr undone))
       (w nil)
       (path nil))
      ((null next)
       (setq path (append path (list (implode w))))
       (append '(make preselect ^feature)
               (list feature)
               '(^for) path
               '(^mom <mom>)))
    (cond ((eq next '<)
           (setq path (append path (list (implode w))))
           (setq w nil))
          (t (setq w (append w (list next)))))))


;;; all-but-last returns everything but the last
;;; element of its argument.

(defun all-but-last (l)
  (reverse (cdr (reverse l))))
```

```
;;; prpr prints the results to the port p.
(defun prpr (pr) ; pretty print an OPS5 production
  (terpri p)
  (princ '|(p | p) (print (cadr pr) p)
  (do ((undone (cdddr pr) (cdr undone))
       (next (caddr pr) (car undone)))
      ((null next) (princ '|)| p) (terpri p))
       (terpri p)
       (or (eq next '-->) (princ '|  | p))
       (print next p)))


;;; just-names removes all special symbols and brackets
;;; from an entry condition, leaving just the names.

(defun just-names (ec)
  (delq '=}-
   (delq '-{=
    (delq '-[
     (delq '--
      (delq ']-
       (delq '-<>
        (flatten ec)))))))))


;;; flatten removes all internal nesting from a list.

(defun flatten (l)
  (cond ((atom l) l)
        ((null l) nil)
        ((atom (car l)) (cons (car l) (flatten (cdr l))))
        (t (append (flatten (car l)) (flatten (cdr l))))))
```

Bibliography

Appelt,D.E. 1982 Planning Natural-Language Utterances to Satisfy Multiple Goals. Ph.D. Thesis, Stanford University. Also Technical Note 259, Stanford Reasearch Institute, Menlo Park.

---. 1983 TELEGRAM: a Grammar Formalism for Language Planning. In: Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe: 595-599.

Barr,A. and Feigenbaum,E.A. Eds., 1981 The Handbook or Artificial Intelligence, Volume I. Pitman, London.

Brachman,R.; Amarel,S.; Engelman,C.; Engelmore,R.; Feigenbaum,E. and Wilkins,D. 1983 What are Expert Systems ? In: Hayes-Roth,F.; Waterman,D. and Lenat,D. Eds., Building Expert Systems. Addison-Wesley, London: 31-58.

Brady,M. and Berwick,R. Eds., 1983 Computational Models of Discourse. M.I.T. Press, Cambridge Mass.

Brownston,L.; Farrell,R.; Kant,E. and Martin,N. 1985 Programming Expert Systems in OPS5. Addison-Wesley, Menlo Park.

Buchanan,B.; Barstow,D.; Bechtal,R.; Bennett,J.; Clancey,W.; Kulikowski,C.; Mitchell,T. and Waterman,D. 1983 Constructing an Expert System. In: Hayes-Roth,F.; Waterman,D. and Lenat,D. Eds., Building Expert Systems. Addison-Wesley, London: 127-167.

Bundy, A. 1983 AI 1 Problem Solving Notes. Occasional Paper No. 30, Department of Artificial Intelligence, University of Edinburgh, Edinburgh.

Chandrasekaran,B. and Mittal,S. 1984 Deep Versus Compiled Knowledge Approaches to Diagnostic Problem Solving. In: Coombs,M. Ed., Developments in Expert Systems. Academic Press, London: 23-34.

Chomsky,N. 1975 Reflections on Language. Fontana, Glasgow.

---. 1980 Rules and Representations. Blackwell, Oxford.

Davey,A. 1978 Discourse Production. Edinburgh University Press, Edinburgh.

Downes,W. 1984 Language and Society. Fontana, London.

Firth,J.R. 1957 A Synopsis of Linguistic Theory (1930-1955). In Palmer,F.R. Ed., Selected Papers of J.R.Firth 1952-1959. Longman, London: 168-205.

Forgey,C.L. 1981 OPS5 User's Manual. CMU-CS-81-135, Carnegie Mellon University, Pittsburgh.

Gaschnig,J.; Klahr,P.; Pople,H.; Shortliffe,E. and Terry,A. 1983 Evaluation of Expert Systems: Issues and Case Studies. In: Hayes-Roth,F.; Waterman,D. and Lenat,D. Eds., Building Expert Systems. Addison-Wesley, London: 241-282.

Grosz,H. and Sidner,C. 1985 Discourse Structure and the Proper Treatment of Interruptions. In: Proceedings of the Ninth International Joint Conference on Artificial Intellegence, Los Angeles: 832-839.

Halliday,M.A.K. 1961 Categories of the Theory of Grammar. Word 17(3): 241-292.

---. 1973 Explorations in the Functions of Language. Edward Arnold, London.

---. 1976a English System Networks. In: Kress,G. Halliday: System and Function in Language. Oxford, London: 101-135.

---. 1976b Modality and Modulation in English. In: Kress,G. op.cit.: 189-213.

---. 1976c Intonation and Meaning. In: Kress,G. op.cit: 214-234.

---. 1978 Language as a Social Semiotic. Edward Arnold, London.

---. 1985 An Introduction to Functional Grammar. Edward Arnold, London.

Halliday,M.A.K. and Hasan,R. 1976 Cohesion in English. Longman, London.

Halliday,M.A.K. and Martin,J.R. Eds., 1981 Readings in Systemic Linguistics. Batsford Academic, London.

Hasling,D.; Clancey,W. and Rennels,G. 1984 Strategic Explanation for a Diagnostic Consultation System. In: Coombs,M. Ed., Developments in Expert Systems. Academic Press, London: 117-133.

Hayes-Roth,F.; Waterman,D. and Lenat,D. Eds., 1983 Building Expert Systems. Addison-Wesley, London.

---. 1983a An Overview of Expert Systems. In: ibid.: 3-29.

Hobbs,J. 1985 Granularity. In: Proceedings of the Ninth International Joint Conference on Aritficial Intelligence. Los Angeles: 432-435.

Hopcroft,J. and Ullman,J. 1969 Formal Languages and their Relation to Automata. Addison-Wesley, Reading, Mass.

Hudson,R.A. 1971 English Complex Sentences. North-Holland, London.

---. 1981 Systemic Generative Grammar. In: Halliday,M.A.K and Martin,J.R. Readings in Systemic Linguistics. Batsford Academic, London: 190-217.

deJoia,A. & Stenton,A. 1980 Terms in Systemic Linguistics. Batsford Academic, London.

Kay,M. 1984 Functional Unification Grammar: a Formalism for Machine Translation. In: Proceedings of COLING84. Stanford: 75-78.

---. 1985 Parsing in Functional Unification Grammar. In: Dowty,D.; Karttunen,L. and Zwicky,A. Natural Language Parsing. Cambridge University Press, London: 251-278.

Kress,G. Ed., 1976 Halliday: System and Function in Language. Oxford, London.

Leech,G.N. 1983 Principles of Pragmatics. Longman, London.

Malinowski,B. 1923 The problem of meaning in primitive languages. Supplement 1 to: Ogden,C.K. and Richards,I.A. The Meaning of Meaning. Kegan Paul, London.

Mann,W. 1985 The anatomy of a systemic choice. Discourse Processes 8(1): 53-74.

Mann,W./Halliday,M.A.K. Systemic Grammar of English, S.G. Clause Systems. From the PENMAN system, Information Sciences Institute, University of Southern California, Marina Del Rey [No date, but this is an early version].

Mann,W. and Matthiessen,C. 1983 Nigel: A Systemic Grammar for Text Generation. RR-83-105, Information Sciences Institute, University of Southern California, Marina Del Rey.

Martin,J.R. 1984 Functional Components in a Grammar: a review of deployable recognition criteria. In: Nottingham Linguistic Circular 13 (specialissue on systemic linguistics). University of Nottingham, Nottingham: 35-71.

McCord,M. 1975 On the Form of a Systemic Grammar. In: Journal of Linguistics 11(2): 195-210.

McDonald,D.D. 1980 Natural Language Production as a Process of Decision-Making under Constraints. Ph.D. Thesis, M.I.T., Cambridge Mass.

---. 1983a Natural Language Generation as a Computational Problem. In: Brady,M. and Berwick,R. Computational Models of Discourse. M.I.T. Press, Cambridge, Mass.

---. 1983b Description Directed Control: its implications for natural language generation. In Cercone,N. Computational Linguistics Pergamon Press, Oxford: 111-129.

McDonald,D.D. and Pustejovsky,J. 1985 Description-Directed Natural Language Generation. In: Proceedings of the Ninth International Joint Conference on Artificial Intellegence. Los Angeles:

799-805.

McKeown,K. 1982 Generating Natural Language Text in Response to Questions about Database Structure. Ph.D. Dissertation, University of Pennsylvania.

---. 1983 Focus constraints on Language Generation. In: Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe: 582-589.

Mishler,E. 1984 The Discourse of Medicine. Ablex, Norwood, New Jersey.

Monaghan,J. 1979 The Neo-Firthian Tradition and its Contribution to General Linguistics. Max Niemeyer Verlag, Tubingen.

Nilsson,N. 1971 Problem-solving Methods in Artificial Intelligence. McGraw-Hill, London.

Patten,T. 1985 A Problem Solving Approach to Generating Text from Systemic Grammars. In: Proceedings of the Second Conference of the European Chapter of the Association for Computational Linguistics. Geneva: 251-257. Also Research Report No. 260, Dept. of Artificial Intelligence, University of Edinburgh, Edinburgh.

Purves,W.K. 1985 A Biologist Looks at Cognitive AI. In: The AI Magazine 6(2): 38-43.

Quirk,R. and Greenbaum,S. 1973 A University Grammar of English. Longman, Hong Kong.

Quirk,R.; Greenbaum,S.; Leech,G. and Svartvik,J. 1973 A Grammar of Contemporary English. Longman, London.

Ritchie,G.D. 1980 Computational Grammar. Harvester, Sussex.

Sacerdoti,E. 1975 A structure for plans and behaviour. Technical Note 109, Stanford Research Institute, Menlo Park.

Smith,B.C. 1978 A Proposal for a Computational Model of Anatomical and Physiological Reasoning. AI Memo 493, M.I.T., Cambridge Mass.

Sothcott,C. 1985 EXPLAN: A System for Describing Plans in English. M.Sc. Dissertation, Dept. of Artificial Intelligence, University of Edinburgh, Edinburgh.

Stefik,M.; Aikins,J.; Balzer,R.; Benoit,J.; Birnbaum,L.; Hayes-Roth,F. and Sacerdoti,E. 1983 The Architecture of Expert Systems. In: Hayes-Roth,F.; Waterman,D. and Lenat,D. Eds., Building Expert Systems. Addison-Wesley, London: 59-86.

Sullivan,M. and Cohen,P. 1985 An Endorsement-Based Plan Recognition

Program. In: Proceedings of the Ninth International Joint Conference on Artificial Intellegence, Los Angeles: 475-479.

Tate, A. 1975 Interacting Goals and their Use. In: Advance Papers of the Fourth International Joint Conference on Artificial Intelligence. Tbilisi: 215-218.

---. 1976 Project Planning using a Hierarchic Non-linear Planner. Reasearch Memo No. 25, Dept. of Artificial Intelligence, University of Edinburgh, Edinburgh.

Thompson,H. 1977 Strategy and Tactics: A Model for Language Production. In: Papers from the Thirteenth Regional Meeting, Chicago Linguistics Society. Chicago: 651-668.

Waltzman,R. 1983 OPS5 Tutorial. Teknowledge Inc. n.p.

Winograd, T. 1972 Understanding Natural Language. Edinburgh University Press, Edinburgh.

---. 1983 Language as a Cognitive Process. Addison-Wesley, London.

von Wright,G.H. 1971 Explanation and Understanding. Routlege Kegan Paul, London.