SYNTACTIC THEORY AND COMPUTATIONAL LINGUISTICS

Allan Ramsay

1987

PITTSBURGH, PENNSYLVANIA 15213 069

1. What is syntax ?
1.1 Syntax in general linguistics

We take syntax to be the study of structural regularities, both within individual languages and across sets of languages. This study has long been of interest in general linguistics, and especially since Chomsky's pioneering work from the late 1950's onwards it has become perhaps the central topic in the field. In the current chapter we begin by comparing the roles of syntax within general linguistics and computional linguistics. We then consider a number of specific theories about the nature of the syntactic rules of natural language. These theories will include ones which have had considerable influence in general linguistics but which, for reasons which will be detailed, have not been taken up within computational linguistics; and ones which have been developed as a direct result of the presence of the computer as a metaphor and as a constraint.

Syntax is taken to refer to the rules which specify which strings of words from the language in question are acceptable and which are not. As such it does not concern itself with the question of which strings of sounds or orthographic symbols are words of the language, or which strings of words constitute sensible or meaningful sentences. Most linguists who are interested in syntax are well aware that there is a strong connection between the form of a sentence and its meaning. Indeed, many of the regularities which syntactic theories try to capture can only be expressed in terms of relations between the meanings of alternative word strings. The notion of active and passive forms, for instance, can only be discussed once it is realised that *The girl fixed the puncture* and *The puncture was fixed by the girl* have closely related meanings. Despite this strong connection between syntax and semantics, work in general linguistics has attempted as far as possible to keep the descriptions of the two systems separate. Syntactic rules are expressed in terms of structural features (e.g. word order, part of speech, inflection). Semantic rules may be based on syntactic forms, but they do not add anything to them. In computational linguistics, as we shall see, this rigid separation of levels of description is frequently broken down.

General linguistics has also followed Chomsky (1965, 1-11) in making a distinction between 'competence' and 'performance'. This distinction is intended to capture the gap between the rules that a person might have about the allowable forms in a language and the processes that they might use for applying those rules in generation or comprehension. To give just one example, the sentence *The sort of things he likes are generally outdoor sports* is not well-formed, as far as the rules of middle-class British English are concerned, since the subject (*The sort of things he likes*) is singular and the main verb (*are*) is plural. Most native English speakers would, on reflection, accept that it is not well-formed. Nonetheless, nobody would regard it as so unacceptable that they could not understand what it meant, and very few people could be confident that they would never utter a sentence that had the same flaw. The competence, i.e. the knowledge in principle, of a typical English speaker includes some rule of subject/verb agreement which would rule out this example. The performance of most such speakers would be liable to allow it, both as something they might say and as something they would be prepared to accept if someone else said it. General linguistics has concentrated on competence, the rules which make up the language, with little emphasis on the processes people use for applying these rules. Becker (1975) argues strongly that to concentrate on competence rather than performance is a profound mistake, an error in belief as to the correct subject matter for linguistics. A number of workers in computational linguistics, or at least in the related applied field of natural language processing, have taken up similar positions. Riesbeck (1978). as part of the 'conceptual dependency* school of natural language processing (Schank 1975), develops a system which performs a task very similar to syntactic analysis but which is described in terms which do not refer to purely structural matters. In so far as the work of Schank and his colleagues constitutes a theory of language, the explicit denial of a purely syntactic component has a major bearing on the discussion of syntax within computational linguistics. It seems appropriate at this point to remark upon the presence within computational linguistics of theories that deny the very existence of an area of study concerned with structural matters; to point to the discussion of the possible architectures for language processing systems in article 32, and of parsing strategies in article 37; and to concentrate, in the current article, on those theories in computational linguistics where the existence of an explicit level of description concerned with syntax is accepted.

1.2 Syntax in computational linguistics

We have argued that the study of syntax in general linguistics is concerned with structural regularities wherever they are to be found in a natural language or in all natural languages. Regularities which contribute to semantic or pragmatic analysis are indeed noted, but general linguistics would also be interested in structural phenomena which seemed to have no function in the interpretation.

In computational linguistics, the emphasis is often slightly different. Syntactic theories within computational linguistics are often, though not always, developed within larger systems for language processing. Within such systems, the function of the syntactic component is taken as being to encode or decode part of the message carried by the text. Clearly any system which is to express a meaning via natural language text or speech, or extract the meaning from it, must recognise the relation between structure and meaning. Syntactic rules which have direct significance for meaning will be more important, on this view of the function of the syntactic component, than ones which do not. Furthermore, the choice of semantic representation for the overall system is likely to have an influence on the form of syntactic representation chosen - where the analysis of meaning is. as in Eliza (Weizenbaum 1966, 36-45), just a matter of reacting to combinations of keywords, the corresponding syntactic theory will be trivial; text generation systems (McDonald 1983). which need to produce forms which reconcile collections of independent choices concerned with a variety of different aspects of the message, often use some version of Halliday's (1985) systemic grammar, in which the various functions of the range of syntactic options are carefully enumerated. The function of the grammar, in terms of how it contributes to the behaviour of the system within which it is to be used, will often influence the form in which it is described.

The connection between theoretical advances in computational linguistics and technical innovations in working systems also means that the study of syntax is inextricably linked with the development of parsing algorithms. The techniques used in natural language understanding systems are discussed in detail in Chapters 36 and 37 of this handbook, and we shall not repeat the discussion here. We will, however, have to make passing reference to facts about them as we cover particular syntactic formalisms and rules. We will also have to accept that in a number of important systems, structural rules of the sort that would normally be included in the syntactic layer of description are integrated with other sorts of rules, so that it is not possible to extract a separable 'theory of syntax'. For each of the treatments of syntax which we discuss below, we will have to refer to one or more of (i) its interaction with some view of semantics and/or pragmatics, (ii) its relation to parsing and/or generation algorithms, (iii) the degree to which it can be seen as an autonomous theory of syntax at all.

2. Simple phrase structure rules

Before we go on to consider particular syntactic models in detail, we will look briefly at the general notion of rewrite rule which underlies the vast majority of approaches to syntax. The simple notion we are going to present here seems to be inadequate in a number of ways, and the theories discussed below are largely attempts to deal with these inadequacies. Before we can go on to see how successful they are, we have to see what it is they are trying to improve on. This introductory discussion may be skipped by readers with any background in linguistics, who will find it very familiar.

The production of both speech and text is a matter of generating sequences of physical stimuli which can be perceived by some other person or persons. In the case of speech these stimuli are sounds, in the case of text they are marks on paper or some similar medium. It is widely accepted that these sequences can be analysed as sequences of things called 'words'; that individual words often have some relation to the world or to ideas in people's minds which can be triggered in a reader/listener's mind when the word is recognised; and that complex relations between such ideas can be encoded by 'structural relations' between the words in some sequence. This chapter is not concerned with what words are or how the sequence of physical stimuli is construed as a string of words. Nor are we concerned here with how words come to have relationships with either the world or with ideas in people's minds. What we need to understand here is, what are structural relations, and to a lesser extent how are they recognised and how do they encode relations between meanings.

The very first point to note about rules describing possible structural relations is that they are generally couched in terms of word classes, rather than individual words. The relations between *old* and *man* in *the old man*, between *red* and *bus* in *a red bus*, and between *hard* and *problems* in *some hard problems* are universally taken to be the same, and to depend on the fact that *old*, *red* and *hard* are all adjectives and *man*, *bus* and *problem* are all nouns. There is substantial disagreement about exactly what this relation is, but very little argument with the claim that it should be described, at least in part, in terms of categories such as 'adjective' and 'noun'. We will take it for granted in all the following discussion that there are word classes and that syntactic rules are couched in terms of them (in the older literature they are sometimes called 'parts of speech', in recent literature they are sometimes called 'lexical categories').

The simplest notion of structural relationship which we see in syntactic theory is just a labelled relation between one word and another. In the phrase *the black cat*, for instance, we might say that *black* is a modifier of *cat* and *the* is a specifier of *cat*. With this very simple notion, there is no implication that one of the words in the relation is somehow more 'important' than the other, nor is there any restriction on how far apart the words in a relation can be or on what other words can appear between them. There have been attempts, such as Hudson's (1984) 'word grammar', to elaborate syntactic theories in terms of collections of binary relations of this sort, but they have not been used much in computational linguistics and we shall not consider them further (but see (Muraki/Ichiyama/Fukumochi 1985) for at least one example of a computational system using a 'dependency grammar' for Japanese).

It is more usual to describe relationships between groups of words, using some elaboration of the idea of 'rewrite rules'. These are equations which say that one group of symbols is equivalent to another, so that any string containing one can be rewritten as a string containing the other. In the simplest form in which such rules are used in syntactic theory, the left hand side of the equation contains a single symbol and the right hand side contains one or more. The symbols that occur in these rules are the names of syntactic categories. The following extremely simple example will illustrate what can and cannot be done with such sets of rules:

S	=	NP VP
NP	=	determiner noun
VP	=	verb NP

A.: Simple phrase structure rules

This set of rules could be used either to describe how a sentence could be made up of a sequence of words belonging to the categories determiner, noun, verb, determiner, noun; or to show that such a sequence could be interpreted as a sentence. Article 37 will consider programs to do these tasks in detail. The remainder of the current article is concerned with elaborations of the notion of rewrite rule which capture phenomena which are hard to describe with the basic form.

3. Syntactic theories

We will cover the syntactic models which have been developed within computational linguistics, or influenced by it, in an order which roughly reflects how much they can be viewed as independent syntactic theories, rather than as parts of some larger view of language. This order should not be taken to indicate that it is right (or wrong) to regard syntax as an area which can be studied in isolation from other aspects of language, nor does it reflect anything much about historical development or about the author's views as to which are most interesting or important. It should, however, enable readers to see more clearly where computational linguistics has close links with general views on syntax and where it is more closely tied to performance models.

3.1 Transformational grammar
3.1.1. Transformational grammar proper

The most influential syntactic theory developed within general linguistics for the past 30 years or so has been Chomsky's theory of 'transformational grammar', first proposed in (Chomksy 1965). Transformational grammar (henceforth TG) has undergone a long process of development since its first appearance. The basic theory extends the simple notion of rewrite rule by allowing in a collection of rules which perform 'transformations'. In most presentations, e.g. (Akmajian/Heny 1975), transformational rules are presented in a format rather like the following rule for the relation between active and passive forms:

```
SD:     NP  aux       verb    NP
        1    2          3     4
        -------------------------
SC:     4   2>be+en     3     by*1
```

B.: passive transformation

Here SD stands for 'structural description', SC for 'structural change'. The structural description corresponds roughly to the left hand side of a rewrite rule. It specifies the form which a sequence of symbols must have if the rule is to apply to it, and it numbers them so they can be referred to later. The structural change describes the form of the structure which may be obtained by applying the rule, i.e. in this case it consists of the NP that was the fourth component of the original structure, followed by the old second component followed by a past tense form of *be*, followed by the old third component, followed by a structure made out of the the word *by* and the old initial NP (the symbols <, >, + and * denote different ways of attaching one tree structure to another - as leftmost or rightmost child, as children of the same existing parent, or as children of a new parent).

Transformations may also refer to 'features'. These are properties which nodes in a tree may possess in addition to their main category. For instance a noun phrase may possess features specifying whether it is singular or plural, whether it is first, second or third person, what gender it is and so on. These features may be given specific values in some transformation, e.g. the rule for 'dative' movement (to account for the relationship between *give it to me* and *give me it)* might specify that the main verb must be bitransitive, as in:

```
SD: verb(trans«bitrans) NP to NP
     1                  2  3  4
---------------------------------
SC:  1+4*               2  0  0
```

C: Dative movement

We will not go further into the details of particular transformations here, since it turns out that TG, although extremely influential in general linguistics, has had comparatively little impact on computational linguistics. To see why this is so. we have to consider the way in which transformations operate. Early TG posits the following relationships between the objects involved in an utterance:

$$\begin{tikzcd}
\text{SURFACE STRUCTURE} \arrow[r] & \text{SOUNDS} \\
\text{DEEP STRUCTURE} \arrow[u] \arrow[r] & \text{MEANING}
\end{tikzcd}$$

D.: Information structures in TG

The deep structure is a phrase structure tree generated according to the rules of some base phrase structure grammar. The rules of this 'base component' simply determine what objects are legitimate phrase structure trees, i.e. what objects can be deep structures. There are a collection of semantic rules, which relate deep structure phrase trees to meanings. To anything which can be both meant and said, there must correspond a deep structure phrase tree. The transformation rules define the relation between deep structure trees and surface structure trees. Any sequence of transformations that can applied in turn to some deep structure tree will generate another tree, which will be a surface tree. There are then phonological rules which relate the leaves of surface structure trees to sequences of sounds.

Inspection of diagrams such as the above suggest at first sight that TG is proposed as a processing model of language understanding or generation. However, if we want to regard the arrows as indicating information flow then a processing model would be expected to look more like the following:

$$\begin{tikzcd}
\text{Understanding} & \text{Generation} \\
\text{sounds} \arrow[d] & \text{sounds} \\
\text{surface structure} \arrow[d] & \text{surface structure} \arrow[u] \\
\text{deep structure} \arrow[d] & \text{deep structure} \arrow[u] \\
\text{meaning} & \text{meaning} \arrow[u]
\end{tikzcd}$$

E.: information flow in process models

In other words, the rules of a processing model would be expected to show how you could work systematically from sounds to meaning or meaning to sounds. The organisation of figure D. expresses constraints between components of the language system, not routes by which information is carried during the generation or analysis phases of language processing. To construct a computational model of generation based on TG, it would be necessary to indicate how the relevant deep structure tree was obtained for any given 'meaning'. This is not generally done - semantic rules indicate how to interpret a deep structure tree, not how to construct one whose meaning matches the meaning you want to express. To construct a computational model of understanding, it would be necessary to show first how to obtain the surface structure tree corresponding to a given sequence of sounds or graphemes, and then how to find a deep structure tree from which this surface structure tree could be derived by a sequence of transformations. TG does indeed provide a collection of functions from the set of deep structure trees to other sets, e.g. to surface structure and thence to sequences of sounds, or to semantic interpretations; but in both generation and comprehension, deep structure trees are intermediate structures which have to be constructed on the basis of information about either meaning or appearance. What would be needed if TG were to be used as the syntactic component of a language processing program would be the INVERSES of the functions that the theory actually provides. As is well known, functions which are easy to compute do not necessarily have inverses which are easy to compute. This is particularly true of attempts to use TG for comprehension. For this we would need to find a function which could apply sequences of inverse transformations to surface structure trees to get deep structure trees. This would be hard enough if we knew in advance how many transformations had been applied in the derivation of the surface structure tree. It is made far harder by the fact that we do not. Even worse, before we can attempt to relate the surface structure to the deep structure we need to find a function which would get surface structure trees from lexical strings. The function which goes from surface structure to lexical string simply enumerates the leaf nodes of the tree. This is a many-to-one mapping, and as such does not even have an inverse.

TG was originally propounded as a descriptive theory of language, with no explicit claims about its relationship to specific processing mechanisms. The theory is thus not critically damaged by arguments about its computational intractability. However, the same arguments can be recast as criticisms about the overall 'power' of the grammar, in the sense that it provides too few restrictions on what might or might not be possible as a grammar for a natural language, and in this form they are damaging to the goals of the theory. Later work has played down the role of transformations like the passive transformation of figure B., and has also tried to constrain their possible forms, by introducing constraints at various levels. Recent presentations of TG include rather general restrictions on the forms of transformations, derived in large part from work by Ross (1986); constraints (known as 'filters') on the form of the surface structure (Chomsky/Lasnik 1977); constrained specifications of the set of features to be considered in syntactic rules (Jackendoff 1977); and semantic constraints on the co-occurrence of various forms of noun phrase, particularly pronominals (Chomsky 1980) and the 'traces' of noun phrases that are left after transformations have shifted or deleted them from their original positions. The general aim of these constraints is to restrict the range of possible grammars, thus giving the theory greater predictive power and perhaps addressing the problem, raised by Wexler and Culicover (1980), that the rules of the original form of TG are unlearnable under reasonable conditions on the data available to the language learner. As a by-product it might happen that the theory will become less computationally awkward, since it was the presence of complex sequences of transformations that made it difficult in the first place. Unfortunately diagrams of the components of the most recent 'government and binding' form of the theory show that it is still very difficult to relate it to notions of unidirectional information flow either from meaning to lexical string or in the other direction.

$$\begin{tikzcd}
\text{FILTERED SURFACE STRUCTURES} \arrow[r] & \text{SOUNDS} \\
\text{CASE MARKED SURFACE STRUCTURES} \arrow[u] \arrow[r] & \text{SEMANTIC REPRESENTATIONS} \\
\text{SURFACE STRUCTURES} \arrow[u] & \\
\text{DEEP STRUCTURE} \arrow[u] & \\
\text{BASE RULES} \arrow[u] &
\end{tikzcd}$$

F. Information structures in late TG.
(after (Radford 1981))

In figure F., the relations between levels above deep structure are generally constraints on the form that entities at lower levels may take, or on the actions that may be performed upon them. In particular, the laws that connect case marked surface structures and semantic representations constrain, via semantic rules, the co-occurrence of noun phrases in the surface structures. There is thus now a link between semantic representation and lexical string, but it is still not in a form in which it is easy to see how a given semantic representation would drive the generation of a lexical string or vice versa.

We see, then, that the form in which TG is cast makes it awkward to use as the syntactic component of a computational model of language processing. There have been some attempts to apply it in its original form, for instance (Petrick 1973), and it is also possible to develop programs which will test proposed sets of transformations with specified base grammars to see whether or not they lead to the generation of unacceptable sentences (Friedman 1969). These, however, are the exception rather than the rule. The major interaction between TG and computational linguistics has been the way in which computational linguistics has demanded that transformations be used in a more constrained and tractable manner, and hence has contributed towards the recent move to restrict the number and scope of application of transformational rules.

3.1.2. Augmented transition networks

Transformational grammar was originally an attempt to capture the relations between active and passive forms, between questions and assertions, and so on. The mechanisms which were invoked to describe these phenomena included the notion of 'moving things around', and the use of 'features*. The conclusion of section 3.1.1 was that TG had not had as much influence within computational linguistics as it had in general linguistics, but this does not mean that the notions of movement and feature sets have not been taken into computational linguistics.

The most notable form in which they have been used has been within the framework known as 'augmented transition networks' (ATNs). For quite some while this form of grammar, introduced in (Woods 1970), completely dominated the view of syntax held by people working within computational linguistics. It now has a number of serious competitors, but is still the form of grammar most widely discussed in textbooks on computer models of natural language processing (e.g. Charniak and McDermott (1985, 206-222), Rich (1983, 315-320). Winograd (1983)). The ATN formalism arises out of a notation known as 'recursive transition networks' (RTNs) which allows context free phrase structure rules, as introduced in section 2. to be expressed in a way that has a natural computational interpretation. Consider the standard rule describing the structure of an English sentence, and the way it might be used for parsing:

S « NP VP

This rule, when interpreted for parsing, can be seen as saying that to see if some string of words makes up a sentence, see if it makes up a noun phrase followed by a verb phrase. We can rephrase that just slightly so that it reads, to see if something is a sentence try to find a noun phrase, and if you find one then try to find a verb phrase. We can develop a notation which captures this idea of find sequences of structures by writing the rule as a network, which is to be crossed, e.g.

```
     start        NP          VP         end
S:——————————>——————————>——————————>—————————>
```

G.: RTN for a simple sentence

The use of network formalisms in language processing systems, particularly for the syntactic component, is discussed in detail in Article 30. and we have no wish to repeat it at length here. The important point here is the influence that the new notation had on how the grammar is viewed. The switch from rewrite rules to transition networks makes comparatively little difference to the expressive power of the grammar. What it does do is provide a strong indication of how the grammar should be interpreted computationally, namely as a series of attempts to cross arcs.

This in turn suggests that a grammar might be written directly as a program in some suitable language, in which the actions of trying to cross an arc were built in as primitives. Some workers have developed languages explicitly for writing grammars as programs, e.g. Winograd (1972), others have simply made use of some existing language, typically LISP. In either case, programs will contain calls to procedures which traverse 'lexical arcs*, i.e. arcs labelled with the names of lexical arcs, and 'recursive arcs', i.e. ones whose label is the name of a non-terminal category. In parsing, a lexical arc can be traversed if the next word in the input is of the appropriate type, whereas a recursive arc can be traversed if some way can be found through the network for the relevant category (again see Chapter 37 for more details on parsing algorithms for recursive transition networks). In all cases, once the grammar is expressed as a program in some language, it is natural to wonder whether the other facilities of the language can be deployed to increase the expressive power of the grammatical formalism.

Recursive transition networks, of the type described above, are no more powerful than phrase structure grammars. It has often been argued that phrase structure grammars are inadequate for providing a concise and comprehensive coverage of the phenomena observed in the grammars of natural languages. Clearly, then, recursive transition networks are also likely to be inadequate as they stand. The introduction of extra notions from programming languages allows the power of the basic formalism to be extended without abandoning the original ideas on which it was based. In particular, recursive transition networks were converted to augmented transition networks by allowing the programs which represented them to include actions which set and tested 'registers'. These registers function in very much the same way as programming language variables. They are named locations where data may be stored. There are some slight differences between the way that ATN registers behave and the way that straightforward local variables behave, but it is clear that the idea of registers was very strongly influenced by the existence of local variables, and in virtually all implementations of ATN grammars registers are implemented by tinkering with the mechanisms by which local variables are implemented (but see (Ramsay 1985 a) for an alternative option).

Registers can be used in two obvious ways. Firstly they can be used for storing, and subsequently testing, the values of features such as NUMBER or PERSON which are subject to agreement constraints. Thus, freely translating the ATN formalism into English, we might have a rule for sentences which looked something like:

(i) Cross a NP arc.
(ii) Set the value of the register NUMBER to be the same as the value it had for the NP.
(iii) Cross a VP arc.
(iv) Check that the value NUMBER has on the VP is the same as its current value.

H.: ATN for a simple sentence

Here steps (i) and (iii) correspond to the arcs in the recursive network for sentences. Step (ii) checks the value that NUMBER .had at the point when the NP arc had been crossed, and copies it to be the current value of NUMBER. We see here that this register has different manifestations for different networks in just the way that a local variable has different manifestations for different procedure calls; but that, unlike a normal local variable, you can access the value that a register had at the point when the embedded network was crossed. Step (iv) finally checks the number agreement between subject and verb, by inspecting the value of NUMBER for the VP and ensuring that it is the same as the recorded value for the subject.

The example above shows one common way of using registers. The other major use of registers is for recording the function, or role, of some * component of the analysis. There might, for instance, be a register called AGENT for recording which of the NPs in a sentence refers to the entity that actually performed the action referred to by the sentence. For active sentences, the AGENT usually corresponds to the SUBJECT, i.e. the NP which immediately precedes the main verb. For passive sentences, the SUBJECT is more usually *seen* as the PATIENT, i.e. it refers to the entity on which the action was performed. ATN registers may be used to capture this notion of the SUBJECT of a passive sentence being the PATIENT 'moved' out of position as follows:

(i) Cross a NP arc.
(ii) Store the NP as the value of the register AGENT.
(iii) Cross a VP arc.
(iv) If the value of the register VOICE for the VP arc was "passive", copy the value of AGENT to PATIENT.

L: ATN for passive sentence

It is clear from this example that ATNs facilitate the integration of semantic and syntactic processing, so that ATN based analyses often blur the distinction between the two levels. The current example, for instance, mentions the roles AGENT and PATIENT, which might be regarded more as semantic than syntactic categories. It is again the fact that the ATN formalism encourages the view of syntactic rules as parts of a program that leads to the mixing of the two levels, and hence perhaps to a different view of the function and nature of syntax.

Registers may be used for a number of other purposes. The most important remaining function for registers is for holding components which have been 'moved' out of position until they are wanted, e.g. for accounting for topicalisation, as in *Him I don't want to meet*, and for relative clauses, as in *The man who the woman who I was talking to used to be married to*. Most ATNs use a special register called the 'HOLD' register for this function. HOLD differs from other registers in behaving more like a 'free' or 'global' variable, in programming language terms, than a local variable, but it can still easily be seen as a consequence of the translation of recursive transition networks into programs. The existence of the HOLD register raises important questions about the relationship between properties of programming languages and the development of the syntactic theory. The HOLD register is used to deal with one of the most difficult questions for any syntactic theory, namely the apparent movement of syntactic structures. It is hard to escape the feeling that this particular solution is directly a consequence of the presence, in a particular family of programming languages, of the notion of a global variable, and that it is not motivated by much analysis of the way movement rules operate. The ATN formalism has contributed to the development of syntactic theory by offering a new view of the role of syntactic rules, and by giving the theoretician new tools, in the form of facilities from programming languages, with which to describe the linguistic facts. It may also have led to some rather irrelevant discussion of the best way to use the HOLD register, when the question that should really be addressed is whether this register is an appropriate way of dealing with movement rules in the first place.

ATNs were developed to provide a computationally tractable alternative to TG. The use of registers, which is very largely imported from the use of variables in ordinary programming languages, allows many of the phenomena described by TG to be captured in ways which are very similar to the TG characterisations. ATNs have been widely and successfully used in practical natural language systems (Woods 1973. 1975). and many facets of TG have been recreated within the ATN paradigm. Until very recently they had few competitors for the role of the syntactic component of a natural language processing system (although, as we shall see, there have always been people who have tried to develop natural language understanding systems with no syntactic component at all). There has. however, always been some disquiet over the fact that there is no restriction on the programming facilities that can be called upon to set and test the values of registers. Since we can call upon any program we like at any point in an ATN. there is no way in which we can restrict the formal power of the formalism to be less than that of a full Turing machine. There is also no way that we can be sure that the apparent structure exhibited by the network is really a good picture of the rules of the grammar, since it is possible to include arbitrary amounts of processing in addition to the basic acts of traversing lexical and non-lexical arcs. The next section describes a number of formalisms influenced by a slightly different programming paradigm which are claimed to suffer less from these problems, but it remains the case that most existing practical programs use some form of ATN for their syntactic component, and that the ATN continues to be a significant candidate for people wishing to build new systems.

3,2 Unification grammars

ATNs extend the simple notion of phrase structure rule by including the use of ideas from programming, especially the use of variables in the guise of registers. The programming ideas that are used in ATNs are very largely those from the tradition of programming which talks in terms of sequences of actions whose results are stored in variables, from which they can later be retrieved and tested. There is a group of programming languages, exemplified by the logic programming language PROLOG, in which the notion of a variable is radically different. This alternative view of variables has been used to develop a number of different ways of extending the basic phrase structure rule format, as described below.

Under this alternative view, a variable gets a value when an attempt is made to 'unify* it with some object. If you try to unify a variable with something which is not itself a variable, the effect is very much the same as for ordinary variables - the object becomes directly associated with the variable, and can be inspected via reference to the variable. If, however, the thing being unified with the variable is itself another variable, all that happens is that it is noted that the two variables must henceforth always refer to the same thing. There are some 'slight complications, to do with unifying variables which have previously been unified with other concrete objects, and with possible circularities which may arise in the process of unification, but the basic idea is simple. Unification of two objects, whether or not they are variables, is a matter of seeing whether it is possible for them both to stand for the same thing, and if so recording that from now on they do in fact do so. This is so even if they are both variables which have not yet been given values to stand for. As soon as one of them is unified with something which does have a value, the note that they must both stand for the same thing means that the other one must also have the same value. This notion of giving variables values by unifying them was introduced in the course of work on automatic theorem proving by Robinson (1965), and is now used as the variable binding mechanism in the programming language PROLOG.

3.2.1, Definite clause grammar

Unification has also been used to provide a constrained way of using variables within phrase structure rules. Work by Pereira and Warren (1980) on implementing an ATN-like grammar in PROLOG led them to a formalism in which the components of phrase structure rules were supplemented by the presence of sets of variables. When this formalism, called 'definite clause grammar' (DCG), is used for parsing, the structure built when a component of the right hand side of a rule is parsed is unified with its specification. Since the specification will contain variables, the unification will record facts about the values of those variables - possibly that they have certain specific values, possibly just that their values must be the same as those of some other variables. The following DCG rule for a sentence shows how the formalism captures the facts about subject-verb agreement in English:

sentence -• np(NUM.PERS). vp(NUM,PERS)

J.: DCG sentence rule

We have here a simple phrase structure rule supplemented by the presence of the variables NUM and PERS (in most dialects of PROLOG variables are marked as words with initial upper case letters). The rule for NPs will be expected to return a structure such as np(singular, first), which can be unified with the expression np(NUM,PERS) to record the fact that NUM should have the value "singular" and PERS the value "first". When the vp rule subsequently returns its result, the values which it provides for NUM and PERS will also have to be "singular" and "first" or the overall application of the rule will fail.

This use of so-called 'logical variables' provides an elegant account of a number of phenomena, without extending the formal power of the grammatical formalism in an unrestricted manner. In particular, it makes it possible to deal neatly with cases where the values of features are unspecified, and it also describes the movements which the ATN uses the HOLD register for.

To see the first, consider the sentence *The sheep are grazing on the far side of the hill.* The initial NP, *the sheep* is specified for person, but not for number. Application of a rule for NPs would return a result like np(_1, third), where the symbol starting with an underscore denotes an unnamed variable. The unification of this with np(NUM, PERS) would simply have noted that NUM had the same value as this unnamed variable. The VP, *are grazing on the far side of the hill*, is specified for number but not person, so a VP rule would return a result like vp(plural, _2). Unifying this with vp(NUM, PERS) would ensure that NUM had the value "plural", which would then be carried over to the variable _1; and that _2 had the same value as the current value of PERS, namely "third".

The use of logical variables to capture the notion of 'unbounded movement' or 'unbounded dependency' dealt with by the HOLD register in ATNs is rather more intricate. Essentially it involves using a variable called SLASH for recording items which have been found 'out of place', and unifying this variable with components of the right hand sides of rules to see if the object which was found out of place can be made use of within the current rule. The following rules illustrate the use of SLASH for these tasks.

```
sentence(SLASH)  →  np,  vp(SLASH).
sentence(false)  →  np,  sentence(np).

vp(SLASH)   →      v, np(SLASH).
np(np) → .
```

K.: SLASH categories in a DCG

In these rules features other than SLASH have been omitted for clarity. The first sentence rule says that a sentence may consist of a NP followed by a VP, with the sentence and the VP having the same value for SLASH. The second says that a sentence with np as the value of SLASH can be turned into one with false as the value of this feature if it is preceded by an np. This is the rule which captures the notion of movement, by seeing the initial noun phrase as something which should have appeared inside the sentence. By constraining the value of SLASH in the sentence component of the right hand side to be "np", it indicates that somewhere in the analysis of this sentence there will be a place where a noun phrase would normally be required, but where on this occasion it is missing. The verb phrase rule simply notes that an ordinary verb phrase consisting of a verb followed by a noun phrase will inherit the value of SLASH from the noun phrase. The noun phrase rule indicates that one way for a noun phrase to have "np" as the value for SLASH is for it to consist, in fact, of nothing at all - "a noun phrase missing a noun phrase is an empty string". There are, of course, other ways for a noun phrase to have "np" as the value for SLASH, for instance by having a post-modifying relative clause or prepositional phrase which has "np" as the value for SLASH. It is the fact that SLASH may be inherited from nested constituents, where the nesting may be arbitrarily deep, that makes it possible to use it for explanations of unbounded dependencies.

It is not easy to write rules which refer to SLASH so that they operate correctly, any more than it is easy to write ATN rules which refer to HOLD. Furthermore, in both cases the presence of the new feature tends to make straightforward parsing algorithms far less efficient than they were before it was introduced. Nonetheless, SLASH has an advantage over HOLD in that it behaves in exactly the same way as any other logical variable, whereas HOLD does not behave just like any other register. The behaviour of SLASH in the original presentation of DCG was not subjected to a more rigorous analysis than the behaviour of HOLD in most presentations of ATNs, but at least it did not require the importation of any more notions from programming languages than were already required for other phenomena.

The DCG formalism makes use of logical variables, then, to perform a number of the tasks for which ATNs used registers with tests and assignments. There remain a number of phenomena, such as the relation between active and passive forms, between questions and assertions, and so on, which cannot be described easily by anything in the basic DCG formalism. The original presentation of DCGs was forced to allow rules to include explicit reference to arbitrary PROLOG program text, in the same way that ATNs allow reference to arbitrary program text. DCGs have restricted the use of arbitrary program text, but they have not eliminated it. The next few sections describe work which, from a variety of starting points, has led to the development of DCG-like formalisms in which the extra complications are described in a principled way, rather than by recourse to ad hoc programs.

3.2.2. Generalised phrase structure grammar

Generalised phrase structure grammar (Gazdar/Klein/Pullum/Sag 1985), extends the notation of simple phrase structure grammar in three directions, as follow:

(i) An ordinary phrase structure rule such as

```
sentence  →  np  vp
```

states that a sentence may be made up of a np followed by a vp. In generalised phrase structure grammar (GPSG) this statement is split into two components. One, the 'immediate dominance' rule, says that a sentence may be made out of a np and a vp - nothing is said about the order in which they are to appear. The other component, the 'linear precedence' rule, says that WHENEVER a rule refers to a np and a vp on its right hand side, the np must precede the vp. Splitting the grammar into two components in this way may seem likely to make it more cumbersome and less perspicuous. The presence of far-reaching global constraints such as the one given mean that the two stage format captures a number of regularities which would otherwise be invisible, and also that the grammars developed tend in fact to be more, not less, compact than grammars developed without this separation.

(ii) The relation between rules dealing with, say, active and passive forms is captured in GPSG by the presence of 'metarules'. These are rules which specify that the existence of a rule fitting some pattern entails the existence of another related rule. Consider the following metarule for passive forms:

```
VP  →  W, NP
==>
VP[PASSIVE] → W, (PP([by])
```

L.: passive metarule in GPSG

This rule says that for every rule which says that a VP is made up of some items plus a NP, there must be a rule which says that a passive VP may be made up of the same set of items, plus optionally a PP whose preposition is *by*. In all metarules, the letter W stands for an arbitrary, possibly empty, collection of items. Note that metarules are defined over immediate dominance rules, so that a metarule like the above does not say anything about the ordering relations between the items matched by W and the NP or PP.

(iii) Finally GPSG attempts to replace the use of arbitrary computations involving the values of of features by a system of 'co-occurrence restrictions' and other general principles for determining the possible values for features. In GPSG, as in a number of other systems which use unification as the method for describing how feature values are assigned, features may have as their values other features (which may have other features which may ...). The mechanisms which generate the allowable values for features are couched entirely in terms of basic logical operators, such as entailment and equivalence, and simple operations on trees of features. The rules which are built up using these mechanisms are indeed complex, as the properties of natural language would lead us to expect; but at least the fundamental components from which they are built are reasonably restricted and well-defined.

GPSG has received considerable attention within computational linguistics, for a number of reasons. Firstly, the formalism does indeed seem to make it possible to write down concise descriptions of a number of facets of the grammar of natural languages - a fact which should make the theory of interest to both general and computational linguistics. It is hard to provide arguments for or against this claim in the present context - there is insufficient space, and the topic does not really lie within computational linguistics anyway - but it certainly needs to be considered. Secondly, it is clear how a grammar specified as a GPSG might be expanded into the form of either a simple phrase structure grammar, for which efficient parsing and generation algorithms exist, or a DCG, for which reasonable parsing and generation algorithms exist. This led to initial hopes that GPSG could easily be used within practical systems, since 'all' that would have to be done would be to expand the grammar and plug it into some existing algorithm. In practice this has proved to be rather an over-optimistic view. A number of people (Thomson 1982, Ramsay 1985 b) have argued that the best way to use GPSG within computational systems is to expand out the metarules, and then to develop parsing/generation algorithms which are explicitly aimed at the resulting formalisms. Article 37 discusses a number of these processing aspects. It is appropiate to note here that the metarules play, in some sense, a similar role to the transformations of TG, in that they are posited as explanations of how different surface forms may be intimately related at some other level. Just as with transformations, the presence of metarules makes processing far harder than it would otherwise be, and recent versions of GPSG include as few of them as possible. It is significant that neither transformations nor metarules are now used to account for passive forms, which are currently taken to be generated from different starting points from the corresponding actives. The current impression is that the performance of systems using GPSG is not markedly better than that of systems using, say, ATNs of similar power and coverage. The third major advantage of the GPSG formalism is that it was developed with a formal compositional semantics firmly in mind. It is clearly to GPSG's credit that the connection between syntax and semantics is recognised as paramount, and that syntactic rules which do not have semantic signficance are treated as anomalies to be explained. Unfortunately few of the systems which do make use of GPSG grammars interpret them in terms of the full semantics. As the performance of GPSG-oriented parsers improves, and the use of the full semantics becomes more widespread, we will see more systems using on GPSG. At present its most compelling claim is as a framework for explaining and describing grammatical phenomena, and as such it may be most appropriate to judge it from the standpoint of general rather than computational linguistics.

3.2.3. Functional unification grammar

Functional unification grammar (Kay 1985) makes use of very similar mechanisms to GPSG, though with slightly different aims. In functional unification grammar (FUG), as in GPSG, there are a collection of phrase structure rules which specify how groups of structures may be combined. Again as in GPSG, the phrase structure rules are split into two components - a description of the objects that can appear on the right hand side of the rules, plus a partial ordering governing the order in which they may appear. The descriptions of individual structures are couched in terms of features and their expected values, with the assumption that comparison of an object with a specification will be done by unifying them. The partial ordering for a rule is specified within the rule, rather than by global linear precedence rules as in GPSG. Furthermore, the ordering within a rule is specified in terms of the roles the structures are to be allocated to, rather than simply in terms of specific categories being required to precede or follow other categories. In FUG, therefore, it is possible to write rules in which objects belonging to the same category appear, and yet to specify that one of them must precede the other. The following pattern illustrates the format of a FUG rule:

```
CAT = S
PATTERN = (... SUBJECT PREDICATOR ...)

SUBJECT = [CAT = NP]
PREDICATOR = [CAT = VERB]
MOOD = DECLARATIVE
```

M.: Subject verb agreement in FUG

In this pattern there are two distinguished features, namely CAT and PATTERN. All rules have an entry for category, and all rules whose entry for this feature is other than a lexical class name have a PATTERN (or possibly several PATTERNs, since the PATTERN is supposed to specify a partial ordering among constituents, and it may not be possible to specify the appropriate order with a single pattern). In the above rule, the fact that the CAT is S indicates that the rule describes a sentence of some sort. The PATTERN specifies the partial ordering by stating that there must be at least two constituents, one of which is the SUBJECT of the sentence and the other the PREDICATOR, and furthermore that the SUBJECT must immediately precede the PREDICATOR.

The other components of the rule specify that the SUBJECT must have NP as its CAT and the PREDICATOR must have VERB as its category, and that the MOOD of the sentence being described is DECLARATIVE. This rule says a certain amount about what a sentence is, without attempting to say everything, for instance without saying anything about voice. Another sentence rule might be written to describe the form of an active complement-taking sentence, as in

```
CAT = S
PATTERN = (... PREDICATOR OBJECT ...)

OBJECT = [CAT = NP]
PREDICATOR = [CAT = [VERB = [VOICE = ACTIVE]
                            [TRANS]]]
VOICE = ACTIVE
```

N.: Active transitive SENTENCE in FUG

Here we have a description of the relation between the PREDICATOR and the OBJECT of an ACTIVE sentence. The PREDICATOR is expected to come immediately before the OBJECT. The PREDICATOR is again just a verb, but now it has been specified as one which has the feature VOICE=ACTIVE, and which also has the feature TRANS, which may or may not have further sub-features. The object is simply required to be something with CAT=NP, and the whole rule describes a sentence with VOICE=ACTIVE.

These two rules are orthogonal to one another. One states some constraints on the items which are to be the SUBJECT and the PREDICATOR of the sentence, the other states constraints between the PREDICATOR and the OBJECT. Both rules could be conflated into a single rule which embodied the same set of constraints, e.g.

```
CAT = S
PATTERN = (... SUBJ PRED OBJECT ...)

SUBJ = [CAT = NP]
OBJECT = [CAT = NP]
PRED = [CAT = [VERB = [VOICE = ACTIVE]
                      [TRANS]]]
MOOD = DECLARATIVE
VOICE = ACTIVE
```

O.: conflated rule in FUG

There is nothing in FUG to say whether it is better to keep the rules separate or combine them. In the above examples, most people would choose to keep the rule describing the relationship between the SUBJECT and the PREDICATOR separate from the rule describing the relationship between PREDICATOR and OBJECT, since the SUBJECT rule is universal (in English) across sentence types, whereas there are a wide variety of forms for the relationship between the PREDICATOR and its objects and complements. FUG should perhaps be seen less as a theory of grammar than as a rather general notation for writing grammars. The distinction is rather fine. A theory of grammar such as GPSG is supposed to be adequate for capturing the regularities of all and only the natural languages, whereas a grammatical notation like FUG is supposed to be adequate for capturing the regularities of natural languages. For FUG to form a theory, rather than a notation, it would probably need to specify more clearly when two rules should be stated separately and when they should be combined. It should then be possible to see what linguistic forms FUG ruled as possibilities for any natural language whatsoever, and hence to regard it as a theory describing the range of possible languages. As it is, it is a reasonably perspicuous notation which enables the grammar writer to separate out phenomena as required. Systems using FUG grammars can make use of a number of well-known parsing techniques (e.g. unification for feature instantiation, chart parsing for recording well-formed substrings). It is, however, significant that rules of FUG provide partial descriptions of the structures they are concerned with, for instance that the rules in figures M. and N. both talk about different constituents of the same structure. This is a departure from the normal structure of rewrite rules, and necessitates considerable reorganisation of the parsing mechanisms. Kay (1985) gives a detailed algebraic analysis of the changes that are involved.

It should finally be noted that FUG permits the presence of semantically motivated features, and that there is indeed no particular distinction between semantic and syntactic features. As such the formalism, like that of the ATN, perhaps leads to a blurring of the distinction between the two levels. It is significant that FUG has been chosen by a number of workers in language generation for the syntactic component (see e.g. Appelt (1985)).

3.2.4. Lexical functional grammar

Lexical functional grammar (Bresnan/ Kaplan 1982, Bresnan 1978) is another variant on the notion that grammars may best be specified in terms of partially ordered sets of constituents, each described by a set of expected feature values. LFG resembles GPSG more than FUG, in that it is intended as a grammar rather than just as a notation. In LFG, as in both the other cases, the grammar has a component consisting of a set of phrase structure rules and a component for generating new rules. LFG is much more closely related to transformational grammar than are FUG or GPSG, and in LFG the phrase structure component corresponds closely to TG's base component. However, in LFG the component which corresponds to TG's transformational component is regarded as applying solely to lexical items, rather than to the trees generated by the base component. In LFG each word has an entry describing the relations it can enter into with other words or structures. The word *write*, for instance, might have entries which specified that it took a subject and an object (as in *he wrote a letter*) or a subject, an object and a second object (as in *he wrote her a letter*). These would be expressed something like

```
write = write(Subject, Object)
write = write(Subject, Object, Object2)
```

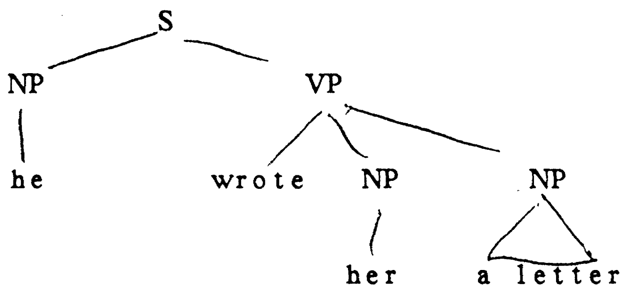
The phrase structure rules have associated with them interpretations in terms of the lexical rules, so that we might have rules like

```
S  →        NP                 VP
      (P^Subject)=↓           P^=↓

VP →    verb             NP
        P^=↓        (P^Object)=↓

VP →    verb             NP                NP
        P^=↓        (P^Object2)=↓    (P^Object)=↓

VP →    verb            NP                PP(to)
        P^=↓        (P^Object)=↓     (P^Object2)=↓
```

P.: Typical LFG rules

The notation here is slightly confusing. An equation like (P^Subject)=↓ means that the item underneath which the equation appears is the subject of the structure described by the whole rule. One like P^=↓ means that the item underneath which it appears is to share all feature values with the structure described by the whole rule, e.g. that in the first rule the VP is to share all features (including Subject) with the S.

The essential point about these rules is that they can be compared directly with the lexical entries to see if and how a given phrase structure rule can be used to analyse a fragment of text containing a particular word. With the entry for *write* given above, and the given set of phrase structure rules, we see that *he wrote her a letter* can be analysed using the second VP rule as a tree roughly like



Q.: LFG parse tree

so long as the sense of *write* corresponding to the second lexical entry is taken, the first NP in the VP is taken to be Object2 and the second NP is taken to be Object. Similar analyses will show that *he wrote a letter to her* is acceptable with the second sense of *write*, so long as *a letter* is taken as Object and *her* as Object2, and *he wrote a letter* is acceptable with the first sense of *write* and *a letter* as Object. Furthermore the annotation of phrase structure rules with functional roles leads to an understanding of how the syntactic forms encode semantic functions. It is important to LFG that the equations attached to phrase structure rules provide a clear and precise semantics, in terms of the mathematical notion of a function. These equations can be used with the lexical entries to filter out inappropriate senses of words and unacceptable applications of phrase structure rules, and hence to constrain the parsing process; to govern the process of semantic interpretation; and to indicate how to choose phrase structure rules to expand functional equations into lexical strings.

LFG originated as an attempt to retain as much as possible of transformational grammar, whilst making parsing and generation more manageable. The biggest change from TG to LFG is the requirement that the 'movement transformations' should be embodied as annotations of the phrase structure rules. This means that there is no need to try arbitrary transformations during the course of analysis or generation. Instead a static set of phrase structure rules are used, with the annotations being used to filter out unacceptable applications of rules. It would be fair to contrast the following diagram of the information structures in LFG with figures D. and F. for TG.

$$\begin{tikzcd}
\text{MEANING REPRESENTATION} \arrow[d] \\
\text{LEXICON} \arrow[d] \\
\text{ANNOTATED PHRASE STRUCTURE RULES} \arrow[d] \\
\text{SURFACE FORMS}
\end{tikzcd}$$

R:. Information structures in LFG

From this diagram we get a clear view of how LFG might be used within a processing framework, and as such the influence of LFG within computational linguistics should come as no surprise. It is, clearly, still necessary to show that LFG does provide a framework within which the real rules of grammar can be specified. Much of the early argument for LFG was concerned to show that it was indeed possible, and even desirable, to replace the transformational component by constraints on the forms associated with lexical items (Bresnan 1978, Bresnan 1982). The reason for LFG's impact on computational linguistics is, as with the other unification grammars, that it provides similar descriptive power to TG without asking computationally unsolved questions about parsing and generation.

All the four unification formalisms we have looked at (DCGs, GPSG, FUG, LFG) have been influenced by the presence of unification as a means of constraining values without overspecifying them ahead of time; and by the existence of efficient algorithms for parsing context free grammars. There has been some progress both in using the formalisms for developing significant grammars for English and other languages, and in employing the parsing and generation techniques with these grammars. It would be premature to suggest that any of the grammars so far developed for any natural language is anywhere near complete; or that the programs which use the grammars we do have are anything like as efficient or robust as we would like. The biggest single problem with all the algorithms for parsing grammars written in these formalisms is to do with the analysis of 'unbounded depndencies" in terms of some category such SLASH (see section 3.2.1). Joshi (1985) presents a formalism called 'tree adjoining grammar' (TAG) which, like GPSG, is intended to capture the same regularities as are covered by TG without extending the power of the grammar much beyond a simple context free language. TAG deals with unbounded dependencies in a way which does not significantly increase the theoretical worst case time for parsing, but the discussion in (Joshi 1985) does not present a concrete parsing algorithm, nor does it deal with the behaviour of other features within the formalism. As such, although TAG seems to have some potential it does not yet seem sufficiently developed to warrant detailed discussion in the present context.

33 Processing systems

The above discussion does not exhaust the range of syntactic theories that have been employed within computational models of language processing. It does, however, cover the majority of purely syntactic theories which have been significantly influenced by the computational paradigm. We will end this article with a consideration of two further purely syntactic models for which this paradigm has indeed been significant, but before that we must remark upon the number of language processing systems which have attempted to deny the very existence of a separable syntactic level. These systems frequently take as their starting point Fillmore's 'case grammar* (Fillmore 1968). It would be inappropriate to go into detail about the components of Fillmore's theory here, since the theory is not purely concerned with syntax, and has anyway not undergone any radical changes as a result of computational influences. It must, however, be noted that large numbers of computational models of language processing have taken the view, with regard to syntax, that purely structural rules are not of interest, and that all analysis should be couched directly in terms of semantic features and functions (see for instance (Hendrix 1977). Wilks (1975. 1977). Reisbeck (1978). Rieger(1975)). If these workers are correct in their analysis, the entire area discussed in this article is an artefact of the linguistic research community, and not something which should be studied at all. The justification for this view is considered in article 32. Its significance for the current discussion is manifest.

3.4 Systemic grammar

Of the remaining syntactic theories to be considered here, systemic grammar (Halliday 1985, Halliday/Martin 1981, Hudson 1971) is another attempt to include some understanding of how rules encode information in a theory of grammar. Systemic grammar is directed especially at understanding how grammatical structure encodes not just the relations between the entities being discussed, but also the relations between these entities and the speaker and listener, and the relations between the speaker, the listener and the state of the discourse. Once it is realised that structural organisation can be used to convey many things at the same time it becomes apparent that a theory of grammar must show not just how each particular thing is encoded, but also how different sets of encodings can be co-ordinated in the same surface string. Halliday (1985) devotes at least as much effort to describing how different forms encode different relationships between clauses, and to how syntax carries information about message structure and discourse, as to the internal structure of a single clause. Foley and Van Valin (1984) also argue that the function of different surface forms is to make it possible to encode different facets of meaning simultanteously.

Systemic grammar is presented in terms of sets of choices to be made by the speaker and recognised by the hearer. Since different choice sets encode different aspects of what is being said, and yet may constrain the form or order of the same words in the output string, the generation or analysis of a particular string may involve problem solving along a number of dimensions. The first, and best known, computational system to use a systemic grammar as its syntactic component is Winograd's SHRDLU system (Winograd 1972). The grammar used in this system was indeed couched at least partly in terms of choice sets, encoded in a special programming language called PROGRAMMAR. Winograd argued strongly in favour of the approach embodied in systemic grammar, namely that grammar carries more information than just the propositional content and mood (i.e. whether a clause is declarative, imperative, a question, or whatever) and that the rules of the grammar should show how different choices contribute to different aspects of meaning. Within Winograd's system, the extra representational power of systemic grammar is used within his theory of 'procedural semantics' (discussed in articles 34 and 38) to allow the semantic component to build into its representations aspects of the function of the input text which are easily extracted from systemic analyses of its structure. There is here, as in FUG, a close connection between the semantic theory chosen for a particular computer implementation and the syntactic theory chosen to support the implementation. Similarly in recent work on language generation (McDonald 1983, Patten 1985) and participation in connected discourse (Sidner 1983, Webber 1983) it is argued that the topics addressed by systemic grammar must be dealt with if computer processing of language is to progress beyond the comprehension of simple isolated sentences. As soon as we start trying to deal with extended texts, either understanding them or generating them, it becomes apparent that syntactic choices do more than just carry propositional content and mood. One of the major messages of computational linguistics for syntactic theory is that the theory must be sensitive to facets of language beyond merely distributional facts about structure, and even beyond facts about the interpretation of sentences in isolation. The notions which systemic grammar is concerned with are becoming more and more important to computational linguistics, though the existing presentations of the theory tend to be rather informal (though comprehensive) and hence in need of considerable reworking before they can actually be used.

3.4.I. Deterministic parsing

We end by considering a theory (Marcus 1980) whose intent is to show that a number of syntactic universals follow directly from the computational properties of a specific parsing algorithm. This strategy is perhaps the most clear-cut way in which computational theories might be used for the development of syntactic theory - if a linguistic property can be shown to be an inevitable consequence of a particular parsing strategy then the parsing strategy can be taken as a tentative *explanation* of the property. Marcus presents a parser which has been carefully constructed so that it can delay making decisions about the type or function of the syntactic structure it is working on until it has enough information to make the right decision. The initial motivation for constructing the parser in this way is the folk-psychological observation that people report a difference between easy to understand sentences and 'garden paths' like the notorious *the horse raced past the barn fell,* for which most people who can understand it all report that they were conscious of having to re-parse it.

Marcus' strategy is to define an architecture for a parsing engine, which is offered as a model of the human parsing mechanism; to develop a particular grammar, in the form of a set of conditionraction rules, which is offered as a subset of the grammar of English, to show that the architecture is at least plausible; and then to argue that a number of Ross' constraints on transformations would be inevitable if this were in fact the architecture which was used. The details of this parser, which is closely related to a class of parsing algorithms known as LL(K) and LR(K) parsers, are discussed in chapter 37. The most significant thing from the point of view of the current article is that the grammar Marcus develops is written in as a set of condition:action rules. This is inevitable, since his predictions about difficulty of parsing are derived precisely from considerations about the amount of storage space available and hence the relative difficulty of delaying a decision on which rule to apply. It unfortunately makes it almost impossible to tell exactly what can be dealt with and what cannot, just as was the case for Riesbeck's conceptual analyser - the details of what can and cannot be parsed by a particular grammar written for Marcus' architecture depends on the detailed timing of when items appear in the various temporary workspaces, which can only be investigated by trying it out. This matters more for Marcus than for Riesbeck. Marcus explicitly wants to show that his program deals with the same range of phenomena as other systems, but that his seldom makes mistakes about how the analysis should proceed. It does build purely structural descriptions, but since the grammar is implicit in the dynamic behaviour of the rule set it is still virtually impossible to see which examples it does cover correctly and which it does not. This work has considerable implications for linguistic theory, in its attempt to show that phenomena which might otherwise be hard to explain follow from computational properties of the parsing mechanism. It has also led to a number of computational models of language acquisition, e.g. (Berwick 1985), which are not open to the criticisms offered by Wexler and Culicover. It does, however, require a new method of assessment, given the difficulty noted above of predicting its coverage of linguistic phenomena. It should also be noted that recent developments in TG have argued that some of Ross' constraints on transformations should be seen as side effects of rules in other parts of the grammar (such as case filters and binding rules). The relationship between developments in TG and changes in Marcus' theory is not clear.

4 Literature (selected)

Akmajian, Adrian/Heny, Frank (1975): *An introduction to the principles of transformational grammar*. Englewood Cliffs, New Jersey.

Appelt, Douglas E. (1985): *Planning English sentences*. Cambridge.

Becker, Joseph D. (1975): The phrasal lexicon. In: *Theoretical advances in natural language processing 1*. Cambridge, Mass., 70-73.

Berwick, Robert C. (1985): *The acquisition of syntactic knowledge*. Cambridge, Mass.

Boguraev, Branimir K. (1979): *Automatic resolution of linguistic ambiguities*. Technical report 11, University of Cambridge Computer Laboratory.

Bresnan, Joan W. (1978): A realistic transformational grammar. In: *Linguistic theory and psychological reality*. Halle, M./Bresnan, J.W./Miller, G.A., eds., Cambridge, Mass., 1-58.

Bresnan, Joan W./Kaplan, Ron (1982): Lexical functional grammar; a formal system for grammatical representation. In: *The mental representation of grammatical relations*. Bresnan, J.W., ed. Cambridge, Mass.

Charniak, Eugene/McDermott, Drew V. (1985): *Introduction to artificial intelligence*. Reading, Mass.

Chomsky, Noam (1965): *Aspects of the theory of syntax*. Cambridge, Mass.

Chomsky, Noam (1981): *Lectures in government and binding*. Dordrecht.

Chomsky, Noam/Lasnik, Howard (1977): Filters and control, *Linguistic Inquiry 8*, 425-504.

Dowty, David R./Wall Robert/Peters, Stanley (1981): *Introduction to Montague semantics*. Dordrecht.

Fillmore, Charles J. (1968): The case for case. In: *Universals in linguistic theory*. Bach, E./Harms, R.T., eds., New York, 1-90.

Foley, William A./Van Valin, Robert D. (1984): *Functional syntax and universal grammar*. Cambridge.

Friedman, Joyce (1969): Directed random generation of sentences. In: *Communications of the Association for Computing Machinery* 12, 40-46.

Gazdar, Gerald/Klein, Ewan/Pullum, Geoffrey K./Sag, Ivan.A (1985): *Generalised phrase structure grammar*. Oxford.

Halliday, M.A.K. (1985): *An introduction to functional grammar*. London.

Halliday, M.A.K./Martin, James R. (1981): *Readings in systemic linguistics*. London.

Hendrix, Gary G. (1977): *The LIFER manual: a guide to building practical natural language interfaces*. Technical note 138, Artificial Intelligence Center, SRI International.

Hudson, Richard A. (1971): *English complex sentences: an introduction to systemic grammar*. Amsterdam.

Hudson, Richard A. (1984): *Word grammar*. Oxford.

Joshi, Aravind (1985): Tree adjoining grammars. In: *Natural language parsing*. Dowty, D.R./Karttunen L./Zwicky A.M., eds., Cambridge, 206-250.

Kay, Martin (1985): Parsing in functional unification grammar. In: *Natural language parsing*. Dowty, D.R./Karttunen L./Zwicky A.M., eds., Cambridge, 251-278.

Marcus, Mitch P. (1980): *A theory of syntactic recognition for natural language*. Cambridge, Mass.

McDonald, David D. (1983): Natural language generation as a computational problem: an introduction. In: *Computational models of discourse*. Brady, J.M./Berwick, R.C., eds., Cambridge, Mass., 209-264.

Montague, Richard (1974): *Formal philosophy*. New Haven.

Muraki, Kazunori/Ichiyama, Shunji/Fukumochi, Yasutomo (1985): Augmented dependency grammar: a simple interface between the grammar rule and the knowledge. In: *Proceedings of 2nd European conference on computational linguistics*, 198-205.

Patten, Terry (1985): A problem solving approach to generating text from systemic grammars. In: *Proceedings of 2nd European conference on computational linguistics*, 251-257.

Pereira, Fernando C.N/Warren, David H.D. (1980): Definite clause grammars for language analysis-a survey of the formalism and a comparison with augmented transition networks. In: *Artificial Intelligence* 13(13), 231-278.

Petrick, Stanley (1973): Transformational analysis. In: *Natural language processing*. Rustin, R., ed., New York, 27-41.

Radford, Andrew (1981): *Transformation syntax*, Cambridge.

Ramsay, Allan M. (1985 a): On efficient context switching. In: *The computer journal* 28(4), 375-378.

Ramsay, Allan M. (1985 b): Effective parsing with generalised phrase structure grammar. In: *Proceedings of 2nd European conference on computational linguistics*, 57-61.

Rich, Elaine (1983): *Artificial Intelligence*. New York.

Rieger, Charles (1975): Conceptual memory and inference. In: *Conceptual information processing*. Schank, R.C., ed., New York, 157-288.

Riesbeck. Christopher K. (1978): An expectation-driven production system for natural language understanding. In: *Pattern directed inference systems.* Waterman, D.AYHayes-Roth. R.. *eds.,* New York. 399-414.
Robinson. J.A (1965): A machine-oriented logic based on the resolution principle. In: Journal of the Association of Computing Machinery. 12(1). 23-41.
Ross, John R. (1986): *Infinite syntax,* Cambridge. Mass.
Schank. Roger C. (1975): The primitive ACTs of conceptual dependency. In: *Theoretical advances in natural language processing 1,* 38-41.
Schank. Roger C./Riesbeck Christopher K. (1981): *Inside computer understanding: five programs plus miniatures.* New Jersey.
Sidner. Candice L. (1983): Focusing in the comprehension of definite anaphora. In: *Computational models of discourse.* Brady. J.M./Berwick. R.C., eds.. Cambridge. Mass., 267-328.
Simmons. Robert F. (1973): Semantic networks: their computation and use for understanding English sentences. In: *Computer models of thought and language.* Schank. R.C./Colby. K.M.. *eds..* San Francisco. 63-113.
Thomson. Henry (1982): *Handling metarules in a parser for GPSG.* Department of Artificial Intelligence Report 175. University of Edinburgh.
Webber. Bonnie L. (1983): So what can we talk about now? In: *Computational models of discourse.* Brady, J.M./Berwick. R.C., *eds.,* Cambridge. Mass.. 331-370.
Weizenbaum. Joseph P. (1966): ELIZA. In: *Communications of the Association for Computing Machinery 9,* 36-45.
Wexler, Kenneth/Culicover. Peter W. (1980): *Formal principles of language acquisition,* Cambridge. Mass.
Wilensky, Robert (1978): Why John married Mary: understanding stories involving recurring goals. In: *Cognitive Science* 2. 235-366.
Wilks. Yorick (1975): A preferential, pattern-seeking semantics for natural language inference. In: *Artificial Intelligence* 6, 53-74.
Wilks, Yorick (1977): Time flies like an arrow. In: *New Scientist* 76. 696-698.
Winograd. Terry (1972): *Understanding natural language.* New York.
Winograd. Terry (1982): *Language as a cognitive process.* Reading. Mass.
Woods. William A. (1970): Transition network grammars for natural language analysis. In: *Communications of the Association for Computing Machinery* 13. 591-606.
Woods. William A. (1973): Progress in natural language understanding: an application to lunar geology. In: *Proceedings of the American federation of information processing societies conference* 42. 441-450.
Woods. William A. (1975): SPEECHLIS: an experimental prototype for speech understanding research. In: *IEEE transactions on acoustics, speech and signal processing* 23(1), 2-10.