49C

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Artificial Intelligence
Memo No. 181

November 1969

PROGRAMMAR:
A Language for Writing Grammars

Terry Winograd

This Memo describes PROGRAMMAR, a parser for natural language. It consists of a language for writing grammars in the form of programs, and an interpreter which can use these grammars to parse sentences. PROGRAMMAR is one part of an integrated system being written for the computer comprehension of natural language. The system will carry on a discourse in English, accepting data statements, answering questions, and carrying out commands. It has a vertically integrated structure, to perform parsing, semantic analysis, and deduction concurrently, and to use the results of each to guide the course of the entire process. This interaction is possible because all three aspects are written in the form of programs. This will allow the system to make full use of its "intelligence" (including non-linguistic knowledge about the subject being discussed) in interpreting the meaning of sentences.

Table of Contents

I A System for Understanding Natural Language

II A Description of PROGRAMMAR

- II.1 Introduction
- II.2 Grammar and Computers
- II.3 Context-free and Context-sensitive grammars
- II.4 Systemic Grammar
- II.5 Grammars as Programs
- II.6 The Form of PROGRAMMAR Grammars
- II.7 Context-sensitive Aspects
- II.8 Ambiguity and Understanding
- II.9 Summary

III Details of the PROGRAMMAR Language

- III.1 Operation of the System
- III.2 Special Words
- III.3 The Dictionary
- III.4 Backup Facilities
- III.5 Messages
- III.6 The Form of the Parsing Tree
- III.7 Variables Maintained by the System
- III.8 Pointers
- III.9 Feature Manipulating

IV Examples of Sentences Parsed

B1

3

I. A SYSTEM FOR UNDERSTANDING NATURAL LANGUAGE

This paper desribes PROGRAMMAR, a language for the writing of grammars. PROGRAMMAR was designed as an integral part of a system for the computer understanding of English. The system answers questions, executes commands, and accepts information in normal English sentences. It uses semantic information and context to understand pronoun references in discourse and to disambiguate sentences both grammatically and semantically. To do this it uses a special type of representation for both the grammar and the semantics. By representing these in the form of programs, it is possible to make full use of semantic and contextual information in developing an analysis of the sentence. It combines a complete grammatical analysis of the sentence with a "heuristic understander" which uses all of its information about the sentence, the discourse, and the world in finding the meaning of a sentence.

The purpose of building this system was not to develop an applied system, but to explore the problems of grammar, semantics, and logic which must be solved for the effective understanding of language.

The project consists of 6 interrelated parts, of which PROGRAMMAR is the first. Roughly they can be described as follows:

1] A system for the grammatical analysis of input sentences. There have been many different parsing systems developed by different language projects, each based on a particular theory of grammar. The type of grammar chosen plays a decisive role in the type of semantic analysis which can be carried out, and PROGRAMMAR was designed specifically to fit the type of analysis used in this system. It differs from other grammars in that the grammar itself is written in the form of a program, and the parsing system is in effect an interpreter for the language used in writing those programs.

2} a grammar of English to be used by the system. I have written a fairly comprehensive grammar of English as a first approximation, following the lines of systemic grammar (see Section II.4). There is one basic criterion for the completeness of the grammar. A person with no knowledge of the system or its grammar should be able to type any reasonable sentence within the limitations of the system's vocabulary and expect it to be understood.

3] A semantic system for exctracting the meaning of the sentence from its grammatical form This again is a combination of a system and a language. There are mechanisms for setting up simple types of semantic networks and using deductions from them as a first phase of semantic anaysis. More important, the meaning of a word or construction is also defined in the form of a program to be interpreted in a semantic language. It is this aspect of semantics

which is missing in most other theories, which limit themselves to a particular type of network or relational structure. Part of this must include a powerful heuristic program for resolving ambiguities and determining the meaning of references in discourse. In almost every sentence, reference is made either explicitly (as with pronouns) or implicitly (as with the word "too") to objects and concepts not explicitly mentioned in that sentence. To interpret these, the program must have at its disposal not only a detailed grammatical analysis (to check for such things as parallel constructions), but also a powerful deductive capacity, and a thorough knowledge of the subject it is discussing.

B

4] A deductive system which can be used by the semantic system in carrying out the deductions which are necesssary not only for such things as resolving ambiguities and answering questions, but also for the directing of the parser. I plan to use PLANNER, a deductive system designed by Carl Hewitt (A.I. Memo 185) which is based on a philosophy very similar to the general mood of this project. Deduction is not carried out by a general procedure acting on a set of axioms or theorems expressed in a formal system of logic. Instead, each theorem is in the form of a program, and the deductive process can be directed by intelligent theorems. PLANNER is actually a language for the writing of those theorems.

5] A generative language capacity to produce answers to questions and to ask questions when necessary to resolve ambiguities. Grammatically this is much less demanding than the interpretive capacity, since humans can be expected to understand a wide range of responses, and it is possible to express almost anything in a grammatically simple way. However, it takes a sophisticated semantic and deductive capability to phrase things in a way which is meaningful and natural in discourse.

6] A base of semantic knowledge. This must include not only "dictionary definitions", but also an understanding of the subject being discussed. This will enable the system to make the deductions needed to uvderstand what is being said. This will include both statements in the "semantics" language (equivalent to the traditional dictionary) and detailed PLANNER statements which include the practical knowledge of the world. I would like to experiment with more than one field of discourse, but the emphasis will be on depth of understanding rather than breadth. It will not try to deal with the entire contents of a childrens' encyclopedia, or to treat objects as formal items and accept statements on all subjects whatsoever. The system will be able to answer deep questions on a particular subject. For example, we might consider a robot with an eye, an arm, and the ability to manipulate simple objects. We would like to be able to say, "Why did you pick up the green block while you were building the tower in the corner?", or "How many blocks were behind the green one when it was the bottom of a three block stack?" (note the use of "one", "it", etc.) and to actually enter into discourse with a sequence of questions, like "Is there a sphere on the table?" "What color is it?" "When was it put there?" "Were there any towers then?", etc. this will require giving the system a detailed model not only of the properties of blocks, hands, tables, etc., but a model of its own mentality as well. It should be able to discuss its plans and thoughts, as well as use them.

Listing these six aspects of language understanding separately is somewhat misleading, as it is the interconnection and interplay between them which makes the system possible. The parser does not parse a sentence, then hand it off to an interpreter. As it finds each component it checks its semantic intepretation, first to see if it is plausible, then if possible to see if it is in accord with the system's knowledge of the world, both specific and general. This has been done in a limited way by other systems, but I plan to use it as an integral part of understanding at every level. The features of PROGRAMMAR are important not only in its effectiveness as a parser, but in the way they lend themselves to use as part of this complete language process.

II. A Description of PROGRAMMAR

II.1 Introduction

PROGRAMMAR differs from other computer systems for natural language in three major ways.

First, it is based on a different type of grammar: systemic grammar, developed by M.A.K. Halliday and others at the University of London (see references 2-5). This is discussed further in Section II.4. The important thing about it is not its specific form or details, but the fact that its general bias is towards seeing language as a system for conveying meaning, not as strings of symbols. It of course does manipulate the symbols of language, but in its analysis it makes heavy use of the very special ways in which language systematically organizes and conveys meaning. This emphasis makes it much more suitable for inclusion in a total understanding system than a more strictly formal theory, such a transformational grammar or any of the variations of context-free grammars.

Second, PROGRAMMAR views a grammar as a program, and places great emphasis on the process of understanding a sentence, rather than the form of the abstract structures underlying it. This distinction does not give an abstract theoretical advantage since any type 0 grammar has the same power. What it does allow is the organization of knowledge about language into a very concise, usable, and efficient form. Most current grammatical theories are expressed in the form of a process, usually a generation process which acts on a set of syntax rules, each of which represents a simple process of replacement. By allowing the grammar to be expressed explicitly in the form of a program, we can express regularities and facts about language which would take very complex 13
additions to more rule-oriented forms of grammar. It also makes it possible

to intermix the grammatical and semantic programs in a more intimate way, allowing them to work together at each point of the process of understanding.

Of course this process is not intended as a direct model of a process used by humans any more than the generation rules of other theories are believed to be actually carried out by a language user. The fact that a program can be the best way to structure and convey knowledge does not imply that the program has a psychological reality. It does, however suggest that this type of representation may be highly useful and revealing in describing human behavior.

Third, PROGRAMMAR sees language as a process of intepretation. A grammar is a program for interpreting sentences, rather than generating them. This approach has has a number of important consequences to the entire process of linguistic understanding . This is discussed at length in another paper (reference 6), and will not be described in detail here. We will note that it makes the integration of the different levels of grammar, semantics and deduction much simpler by giving us a coherent way to decide at which of these levels each type of linguistic fact is best handled. Many of the problems in current linguistic theories result from trying to explain semantic facts at the syntactic level. By putting syntax into an integrated system, and by allowing semantics to play a real part in sentence interpretation, many facts about language become much more tractable.

II.2 Grammar and Computers

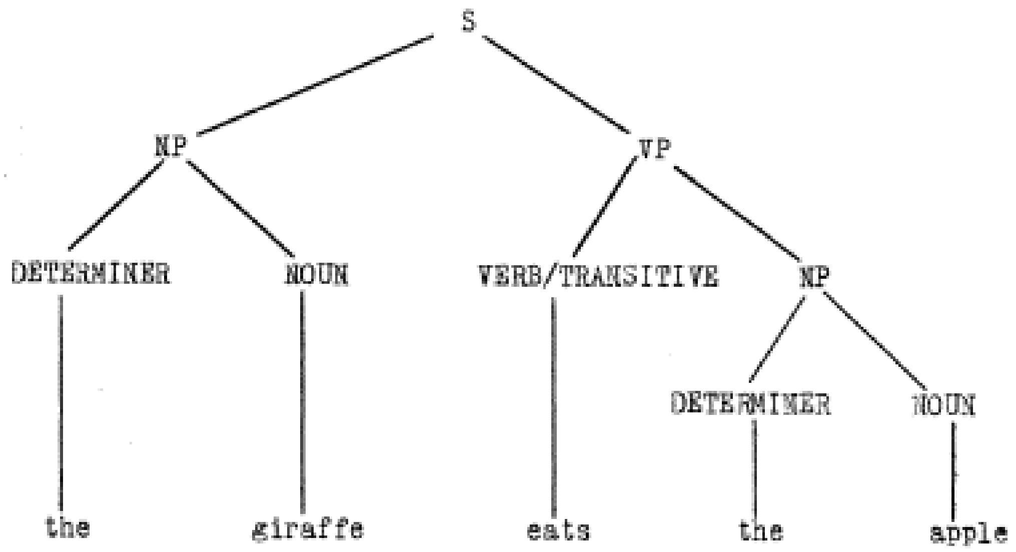
In order to explain the features of PROGRAMMAR, we will summarize some of the principles of grammar used in computer language processing. The basic form of most grammars is a list (ordered or unordered) of "replacement rules," which represent a processs of sentence generation. Each rule states that a certain string of symbols (its left side) can be replaced by a different set of symbols (its right side). These symbols include both the actual symbols of the language (called terminal symbols) and additional "non-terminal" symbols. One non-terminal symbol is designated as a starting symbol, and a string of terminal symbols is a sentence if and only if it can be derived from the starting symbol through successive application of the rules. For example we can write Grammar 1:

```
1.1  S -> NP VP
1.2  NP -> DETERMINER NOUN
1.3  VP -> VERB/INTRANSITIVE
1.4  VP -> VERB/TRANSITIVE NP
1.5  DETERMINER -> the
1.6  NOUN -> giraffe
1.7  NOUN -> apple
1.8  VERB/INTRANSITIVE -> dreams
1.9  VERB/TRANSITIVE -> eats
```

By starting with S and applying the list of rules [1.1 1.2 1.5 1.6 1.4 1.2 1.7 1.5 1.9], we get the sentence "The giraffe eats the apple." Several things are noteworthy here. This is an unordered set of rules. Each rule can be applied any number of times at any point in the derivation where the symbol appears. In addition, each rule is optional. We could just as well have reversed the applications of 1.6 and 1.7 to get "The apple eats the giraffe.", or have used 1.3 and 1.8 to get "The giraffe dreams." This type of derivation

13

can be represented graphically as:



We will call this the parsing tree for the sentence, and use the usual terminology for trees (node, subtreee, daughter, parent, etc.). In addition we will use the linguistic terms "phrase" and "constituent" interchangeably to refer to a subtree. This tree represents the "immediate constituent" structure of the sentence.

II.3 Context-free and Context-sensitive Grammars

Grammar 1 is an example of what is called a context-free grammar. The left side of each rule consists of a single symbol, and the indicated replacement can occur whenever that symbol is encountered. There are a great number of different forms of grammar which can be shown to be equivalent to this one, in that they can characterize the same languages. It has been pointed out that they are not theoretically capable of expressing the rules of English, to produce such sentences as, "John, Sidney, and Chan ordered an eggroll, a ham sandwich, and a bagel respectively." Much more important, even though they could theoretically handle the bulk of the English language, they cannot do this at all efficiently. Consider the simple problem of subject-verb agreement. We would like a grammar which generates "The giraffe dreams." and "The giraffes dream.", but not "The giraffe dream." or "The giraffes dreams.". In a context-free grammar, we can do this by introducing two starting symbols, S/PL and S/SG for plural and singular respectively, then duplicating each rule to match. For example,we would have:

```
1.1.1  S/PL -> NG/PL VP/PL
1.1.2  S/SG -> NG/SG  VP/SG
1.2.1  NG/PL ->  DETERMINER  NOUN/PL
1.2.2  NG/SG ->  DETERMINER  NOUN/SG
 ...
1.6.1  NOUN/PL ->  giraffes
1.6.2  NOUN/SG ->  giraffe

etc.
```

If we then wish to handle the difference between "I am", "he is", etc. we must introduce an entire new set of symbols for first-person. This sort of duplication propagates multiplicatively through the grammar, and arises in all sorts of cases. For example, a question and the corresponding statement will have much in common concerning their subjects, objects verbs, etc., but in a context-free grammar, they will in general be expanded through two entirely different sets of symbols.

One way to avoid this problem is to use context-sensitve rules. In these, the left side may include several symbols, and the replacement occurs when that combination of symbols occurs in the string being generated.

II.4 Systemic Grammar

We can add power to our grammar with context-sensitve rules which, for example, in expanding the symbol VERB/INTRANSITIVE, look to the preceding symbol to decide whether it is singular or plural. By using such context-sensitive rules, we can characterize any language whose sentences can be listed by a deterministic (possibly neverending) process. (i.e. they have the power of a turing machine). There is however a problem in implementing these rules. In any but the simplest cases, the context will not be as obvious as in the simple example given. The choice of replacements will not depend on a single word, but may depend in a complex way on the entire structure of the sentence. Such dependencies cannot be expressed in our simple rule format, and new types of rules must be developed. Transformational grammar solves this by breaking the generation process down into the context-free base grammar which produces "deep structure" and a set of transformations which then operate on this structure to produce the actual "surface structure" of the grammatical sentence. We will not go into the details of transformational grammar, but one basic idea is this separation of the complex aspects of language into a separate transformational phase of the generation process.

Systemic grammar introduces context in a more unified way into the immediate-constituent generation rules. This is done by introducing "features" associated with constituents at every level of the parsing tree.

A rule of the grammar may depend, for example, on whether a particular clause is transitive or intransitive. In the examples "Fred found a frog.","A frog was found by fred.", and "What did fred find?", all are transitive, but the outward forms are quite different. A context-sensitive rule which checked for this feature directly in the string being generated would have to be quite complex. Instead, we can allow each symbol to have additional subscripts, or features which control its expansion. In a way, this is like the separation of the symbol NP into NP/PL and NP/SG in our augemented context-free grammar. But it is not necessary to develop whole new sets of symbols with a set of expansions for each. A symbol such as CLAUSE may be associated with a whole set of features (such as TRANSITIVE, INTERROGATIVE, SUBJUNCTIVE, OBJECT-QUESTION, etc.) but there is a single set of rules for expanding CLAUSE. These rules may at various points depend on the set of features present.

The power of systemic grammar rests on the observation that the context-dependency of natural language is centered around clearly defined and highly structured sets of features, so through their use a great deal of complexity can be handled very economically. More important for our purposes, there is a high correlation between these features and the semantic interpretation of the constituents which exhibit them. They are not directly semantic, but are a tremendous aid to interpretation. A parsing of a sentence in a systemic grammar might look very much like a context-free parsing tree, except that to each node would be attached a number of features.

These features are not random combinations of facts about the constituent, but are a part of a carefully worked out analysis of a language in terms of its "systems". The features are organized in a network, with clearly organized dependencies. For example, the features IMPERATIVE (command) and INTERROGATIVE (question) are mutually exclusive in a clause, as are the features POLAR (yes-no question like "Did he go?") and WH-QUESTION (like "Who went?). In addition, the second choice can be made only if the choice INTERROGATIVE was made in the first set. A set of mutually exclusive features is called a "system", and the set of other features which must be present for the choice to be possible is called the "entry condition" for that system. This is discussed in detail in references 4 and 5.

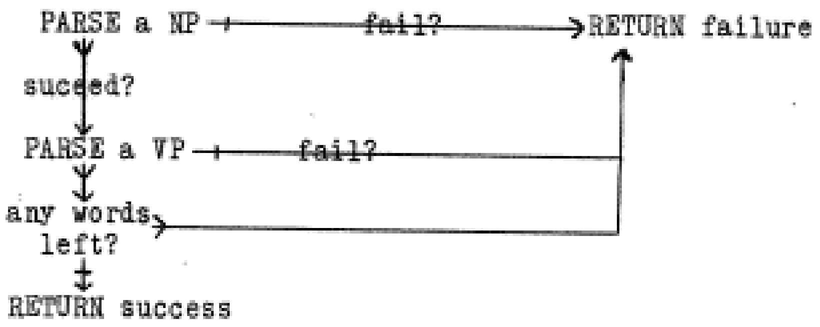
Another basic concept of systemic grammar is that of the rank of a constituent. Rather than having a plethora of different non-terminal symbols, each expanding a constituent in a slightly different way, there are only a few basic "units", each having the possibility of a number of different features, chosen from the "system network" for that unit. In an analysis of English, three basic units seem to explain the structure: the CLAUSE, the GROUP, and the WORD. In general, clauses are made up of groups, and groups made up of words. However, through "rankshift", clauses or groups can serve as constituents of other clauses or groups. Thus, in the sentence "Sarah saw the student sawing logs." "the student sawing logs" is a NOUN GROUP with the CLAUSE "sawing logs" as a constituent (a modifier of "student").

The constituents "who", "three days", "some of the men on the board of directors," and "anyone who doesn't understand me" are all noun groups, exhibiting different features. This means that a PROGRAMMAR grammar will have only a few programs, one to deal with each of the basic units. My current grammar of English has programs for the units CLAUSE, NOUN GROUP, VERB GROUP, PREPOSITIONAL GROUP, and ADJECTIVE GROUP.

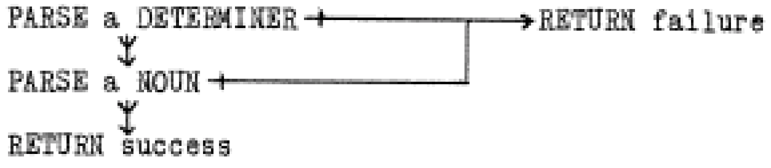
II.5 Grammars as Programs

Let us see how a grammar as outlined above could be written as a program. Grammar 1 could be diagrammed:

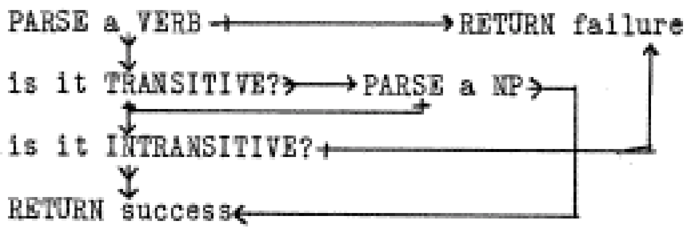
DEFINE program SENTENCE



DEFINE program NP



DEFINE program VP



The basic function used is PARSE, a function which tries to add a constituent of the specified type to the parsing tree. If the type has been defined as a PROGRAMMAR program, PARSE activates the program for that unit, giving it as input the part of the sentence yet to be parsed. If no definition exists, PARSE interprets its arguments as a list of features which must be found in the dictionary deinition of the next word in the sentence. If so, it attaches a node for that word, and removes it from the remainder of the sentence. If not, it fails. If a PROGRAMMAR program has been called and succeeds, the new node is attached to the parsing tree. If it fails, the tree is left unchanged.

P

II.6 The Form of PROGRAMMAR Grammars

Written in PROGRAMMAR, the programs would look like:

```
2.1  (PDEFINE SENTENCE
2.2  (((PARSE NP) NIL FAIL)
2.3   ((PARSE VP) FAIL FAIL RETURN)))

2.4  (PDEFINE NP
2.5  (((PARSE DETERMINER) NIL FAIL)
2.6   ((PARSE NOUN) RETURN FAIL)))

2.7  (PDEFINE VP
2.8  (((PARSE VERB) NIL FAIL)
2.9   ((ISQ H TRANSITIVE) NIL INTRANS )
2.10  ((PARSE NP) RETURN NIL)
2.11 INTRANS
2.12  ((ISQ H INTRANSITIVE) RETURN FAIL)))
```

Rules 1.6 to 1.9 would have the form:

```
2.13 (DEFPROP GIRAFFE (NOUN) WORD)
2.14 (DEFPROP DREAM (VERB INTRANSITIVE) WORD)
```
etc.

This example illustrates some of the basic features of PROGRAMMAR. First it is embedded in LISP, and much of its syntax is LISP syntax. Units, such as SENTENCE are defined as PROGRAMMAR programs of no arguments. Each tries to parse the string of words left to be parsed in the sentence. The exact form of this input string is described in section III.6. The value of (PARSE SENTENCE) will be a list structure corresponding to the parsing tree for the complete sentence.

Each time a call is made to the function PARSE, the system begins to build a new node on the tree. Since PROGRAMMAR programs can call each other recursively, it is necessary to keep a pushdown list of nodes which are not yet completed (i.e. the entire rightmost branch of the tree). These are all called "active" nodes, and the one formed by the most recent call to PARSE is called the "currently active node".

We can examine our sample program to see the basic operation of the language. Whenever a PROGRAMMAR program is called directly by the user, a node of the tree structure is set up, and a set of special variables are bound (see section III.7). The lines of the program are then executed in sequence, as in a LISP PROG, except when they have the special form of a BRANCH statement, a list whose first member (the CONDITION) is non-atomic, and which has either 2 or 3 other members, called DIRECTIONS. Line 2.3 is a three-direction branch, and all the other executable lines of the program are two-direction branches.

When a branch statement is encountered, the condition is evaluated, and branching depends on its value. In a two-direction branch, the first direction is taken if it evaluates to non-nil, the second direction if it is nil. In a three-direction branch, the first direction is taken only if the condition is non-nil, and there is more of the sentence to be parsed. If no more of the sentence remains, the third direction is taken.

The directions can be of three types. First, there are three reserved words, NIL, RETURN, and FAIL. A direction of NIL sends evaluation to the next statement in the program. FAIL causes the program to return NIL after restoring the sentence and the parsing tree to their state before that program was called. RETURN causes the program to attach the currently active node to the completed parsing tree and return the subtree below that node as its value.

If the direction is any other atom, it acts as a GO statement, transferring evaluation to the statement immediately following the occurence of that atom as a tag. For example, if a failure occurs in line 2.9, evaluation continues with line 2.12. If the direction is non-atomic, the result is the same as a FAIL, but the direction is put on a special failure message list, so the calling program can see the reason for failure.

6

Looking at the programs, we see that SENTENCE will succeed only if it first finds a NP, then finds a VP which uses up the rest of the sentence. In the program VP, we see that the first branch statement checks to see whether the next word is a verb. Af so, it removes it from the remaining sentence, and goes on. If not, VP fails. The second statement uses the PROGRAMMAR function ISQ, one of the functions used for checking features. (ISQ A B) checks to see whether the node or word pointed to by A has the feature B. H is one of a number of special variables used to hold information associated with a node of the parsing tree. (see section III.7) It points to the last word or constituent parsed by that program. Thus the condition (ISQ H TRANSITIVE) succeeds only if the verb just found by PARSE has the feature TRANSITIVE. If so, the direction NIL sends it on to the next statement to look for a NP, and if it finds one it returns success. If either no such NP is found or the verb is not TRANSITIVE, control goes to the tag INTRANS, and if the verb is INTRANSITIVE, the program VP succeeds. Note that a verb can have both the features INTRANSITIVE and TRANSITIVE, and the parsing will then depend on whether or not an object NP is found.

B

II.7 The Context-Sensitive Aspects

So far, we have done little to go beyond a context-free grammar. How, for example, can we handle agreement? One way to do this would be for the VP program to look back in the sentence for the subject, and check its agreement with the verb before going on. We need a way to climb around on the parsing tree, looking at its structure. In PROGRAMMAR, this is done with the pointer PT and the moving function *.

Whenever the function * is called, its arguments form a list of instructions for moving PT from its present position. The instruction list contains non-atomic CONDITIONS and atomic INSTRUCTIONS. The instructions are taken in order, and when a condition is encountered, the preceding instruction is evaluated repeatedly until the condition is satisfied. If the condition is of the form (ATOM), it is satisfied only if the node pointed to by PT has the feature ATOM. Any other condition is evaluated by LISP, and is satisfied if it returns a non-nil value. Section III.8 lists the instructions for *.

For example, evaluating (* C U) will set the pointer to the parent of the currently active node. (The mnemonics are: Current, Up) The call (* C DLC PV (NP)) will start at the current node, move down to the rightmost completed node (i.e. not currently active) then move left until it finds a node with the feature NP. (Down-Last-Completed, PreVious). If * succeeds, it returns the new value of PT and leaves PT set to that value. If it fails at any point in the list, because the existing tree structure makes a command impossible, or because a condition cannot be satisfied, PT is left at its original position, and * returns nil.

We can now add another branch statement to the VP program in section II.6 between lines 2.8 and 2.9 as follows:

```
2.8.1  ((OR(AND(ISQ(* C PV DLC)SINGULAR)(ISQ H SINGULAR))
2.8.2      (AND(ISQ PT PLURAL)(ISQ H PLURAL)))
2.8.3   NIL (AGREEMENT))
```

This is an example of a branch statement with an error message. It moves the pointer from the currently active node (the VP) to the previous node (the NP) and down to its last contituent (the noun). It then checks to see whether this shares the feature SINGULAR with the last constituent parsed by VP (the verb). If not it checks to see whether they share the feature PLURAL. Notice that once PT has been set by *, it remains at that position. If agreement is found, evaluation continues as before with line 2.9. If not, the program VP fails with the message (AGREEMENT).

So far we have not made much use of features, except on words. As the grammar gets more complex, they become much more important. As a simple example, we may wish to augment our grammar to accept the noun groups

"these fish,"

"this fish,"

"the giraffes,"

and "the giraffe,"

But not "these giraffe," or "this giraffes." We can no longer check a single word for agreement, since "fish" gives no clue to number in the first two, while "the" gives no clue in the third and fourth. Number is a feature of the entire noun group, and we must interpret it in some cases from the form of the noun, and in others from the form of the determiner.

We can rewrite our programs to handle this complexity as shown in Grammar 3:

```
3.1  (PDEFINE SENTENCE
3.2  (((PARSE NP)NIL FAIL)
3.3   ((PARSE VP) FAIL FAIL RETURN)))
```

B

```
3.4  (PDEFINE NP
3.5  (((AND(PARSE DETERMINER)(FQ DETERMINED))NIL NIL FAIL)
3.6   ((PARSE NOUN)NIL FAIL)
3.7   ((CQ DETERMINED)DET NIL)
3.8   ((TRNSF H (QUOTE(SINGULAR PLURAL)))RETURN FAIL)
3.9  DET
3.10   ((TRNSF H (MEET(FE(* H PV (DETERMINER)))
3.11                 (QUOTE(SINGULAR PLURAL))))
3.12   RETURN
3.13   FAIL)))

3.14 (PDEFINE VP
3.15 (((PARSE VERB)NIL FAIL)
3.16  ((MEET(FE H)(FE(* C PV (NP)))(QUOTE(SINGULAR PLURAL)))
3.17   NIL
3.18   (AGREEMENT))
3.19  ((ISQ H TRANSITIVE)NIL INTRANS)
3.20  ((PARSE NP)RETURN NIL)
3.21 ((ISQ H INTRANSITIVE)RETURN FAIL)))
```

We have used the PROGRAMMAR functions FQ and TRNSF, which attach features to constituents. The effect of evaluating (FQ A) is to add the feature A to the list of features for the currently active node of the parsing tree. TRNSF is used to transfer features from another node to the currently active node. Its first argument is a pointer to the node from which information is to be transferred. The second is a list of features to be looked for. For example, line 3.8 looks for the features SINGULAR and PLURAL in the last constituent parsed (the NOUN), and adds whichever ones it finds to the currently active node. The branch statement beginning with line 3.10 is more complex. The function * finds the DETERMINER of the NP being parsed. The function FE finds the list of features of this node, and the function MEET intersects this with the list of features (SINGULAR PLURAL). This intersection is then the set of allowable features to be transferred to the NP node from the NOUN. Therefore if there is no agreement beween the NOUN and the DETERMINER, TRNSF fails to find any features to transfer, and the resulting failure causes the rejection of such phrases as "these giraffe."

In line 3.7 we use the function CQ which checks for features on the current node. (CQ DETERMINED) will be non-nil only if the current node has the feature DETERMINED. (i.e. it was put there in line 3.5) Therefore, a noun group with a determiner is marked with the feature DETERMINED, and is also given features corrresponding to the intersection of the number features associated with the determiner if there is one, and the noun. Notice that this grammar can accept noun groups without determiners, as in "Giraffes eat apples." since line 3.5 fails only if a DETERMINER is found and there are no more words in the sentence.

In conjunction with the change to the NP program, the VP program must be modified to check with the NP for agreement.
The branch statement beginning on Line 3.16 does this by making sure there is a number feature common to both the subject and the verb.

This brief description explains some of the basic features of PROGRAMMAR. In a simple grammar, their importance is not obvious, and indeed there seem to be easier ways to achieve the same effect. As grammars become more complex, the special aspects of PROGRAMMAR become more and more important, and I plan in another paper to describe the PROGRAMMAR grammar for Engish which is used by the understander system.

A number of the other features and details of PROGRAMMAR are described in Section III.

B

II.8 Ambiguity and Understanding

Readers familiar with parsing systems may by now have wondered about the problem of ambiguity. As explained, a PROGRAMMAR program tries to find a possible parsing for a sentence, and as soon as it succeeds, it returns its answer. This is not a defect of the system, but an active part of the concept of language for which it was designed. The language process is not segmented into the operation of a parser, followed by the operation of a semantic interpreter. Rather, the process is unified, with the results of semantic interpretation being used to guide the parsing. This is very difficult in other forms of grammar, with their restricted types of context-dependence. But it is straightforward to implement in PROGRAMMAR. For example, the last statement in a program for NP may be a call to a noun-phrase semantic interpreter. If it is impossible to interpret the phrase as it is found, the parsing is immediately redirected.

The way of treating ambiguity is not through listing all 1,243 possible interpretations of a sentence, but in being intelligent in looking for the first one, and being even more intelligent in looking for the next one if that fails. There is no automatic backup mechanism in PROGRAMMAR, because blind automatic backup is tremendously inefficent. A good PROGRAMMAR program will check itself when a failure occurs, and based on the structures it has seen and the reasons for the failure, it will decide specifically what should be tried next. This is the reason for internal failure-messsages, and there are facilities for performing the specific backup steps necessary. (See section III.4)

As a concrete example, we might have the sentence "I rode down the street in a car." At a certain point in the parsing, the NP program may come up with the constituent "the street in a car". Before going on, the semantic analyzer will reject the phrase "in a car" as a possible modifier of "street", and the program will attach it instead as a modifier of the action represented by the sentence. Since the semantic system is a part of a complete deductive understander, with a definite world-model, the semantic evaluation which guides parsing can include both general knowledge (cars don't contain streets) and specific knowledge (Melvin owns a red car, for example). Humans take advantage of this sort of knowledge in their understanding of language, and it has been pointed out by a number of linguists and computer scientists that good computer handling of language will not be possible unless computers can do so as well.

Very few sentences seem ambiguous to humans when they first hear them. They are guided by an understanding of what is said to pick a single parsing and a very few different meanings. By using this same knowledge to guide its parsing, a computer understanding system could take advantage of the same techvique to parse meaningful sentences quickly and efficiently. We must be careful to distinguish between grammatical and semantic ambiguity. Although we want to choose a single parsing without considering the alternatives simultaneously, we want to handle semantic ambiguity very differently. There may be several interpretations of a sentence which are all more or less meaningful, and the choice between them will depend on a complex evaluation of our knowledge of the world, of the knowledge the person speaking has of the world, and of what has been said recently. This is particularly true in cases of ambiguous pronoun reference. For example, if the system were asked the sequence of questions: "Is a green block on a table?" "What color is it?" we would expect "it" to refer to the table. If asked "Is a block on a green table? "What color is it?", we know that "it" must refer to the block. If the second question were "What size is it?" it would be much more ambiguous. To resolve this, we must know that green is a color, and that a person is not likely to ask the color of an object if he has just specified it. This is the type of

B

ambiguity dealt with by the system through its heuristic programs and deductive system. It however is not directly a part of the grammar.

This means that we must discuss only those things the computer knows about and understands fully enough to manipulate the knowledge necessary for interpreting a sentence. The forms of relationships it must be able to handle go beyond simple structures, such as associative networks of links between words or arbitrary relational statments in the predicate calculus. Just how much is necessary is a deep problem, and I plan to discuss it more in reference to other parts of the system.

II.9 Summary

In understanding the reason for developing PROGRAMMAR, several factors are important. The first is that only through the flexibility of expressing a grammar as a program can we introduce the type of intelligence necessary for complete language understanding. PROGRAMMAR is able to take into account the fact that language is structured in order to convey meaning, and that our parsing of sentences depends intimately on our understanding that meaning. PROGRAMMAR can take advantage of this to deal more efficiently with natural language than a general rule-based system, whether context-free or transformational. More important, the analysis returned by PROGRAMMAR is designed to serve as a part of a total understanding process, and to lend itself directly to semantic interpretation. This was one reason for selecting systemic grammar, and has guided much of the design of the system. The exact way in which semantic interpretation can be done, and the reasons why a systemic analysis is important will be discussed in later papers on the semantic aspects of the system.

A reasonably comprehensive grammar of English has been written in PROGRAMMAR, and it is able to parse quite complex sentences. It is being used as a part of the system, which is currently being debugged.

Some examples of English sentences parsed by PROGRAMMAR are included in Section IV.

B

III Details of the PROGRAMMAR Language

III.1 Operation of the System

Since the grammar is itself a program, there is not much overhead mechanism needed for the basic operation of the parser. Instead, the system consists mostly of special functions to be used by the grammar. The system maintains a number of global variables, and keeps track of the parsing tree as it is built by the maiw function, PARSE. When the function PARSE is called for a UNIT which has been defined as a PROGRAMMAR program, the system collects information about the currently active node, and saves it on a pushdown list. It then sets up the necessary variables to establish a new active node, and passes control to the PROGRAMMAR program for the appropriate unit. If this program succeeds, the system attaches the new node to the tree, and returns control to the node on the top of the PDL. If it fails, it restores the tree to its state before the program was called, then returns control. A PROGRAMMAR program is actually converted by a simple compiler to a LISP program and run in that form. The variables and functions available for writing PROGRAMMAR programs are described in the rest of part III. Sections III.1 to III.5 explain special features of the language. Sections III.6 to III.9 are more in the style of a manual which would allow the reader to understand PROGRAMMAR programs.

When the function PARSE is called with a first argument which has not been defined as a PROGRAMMAR program, it checks to see whether the next word has all of the features listed in the arguments. If so, it forms a new node pointing to that word, with a list of features which is the intersection of the list of features for that word with the allowable features for the word class indicated by the first argument of the call. For example, the word "blocks" will have the

B

possibility of being either a plural noun or a third-person-singular present-tense verb. Therefore, before any parsing it will have the features (NOUN VERB N-PL VB-3PS TRANSITIVE PRESENT). If the expression (PARSE VERB TRANSITIVE) is evaluated when "blocks" is the next word in the sentence to be parsed, the feature list of the resulting node will be the intersection of this combined list with the list of allowable features for the word-class VERB. If we have defined:

(DEFPROP VERB (VERB INTRANSITIVE TRANSITIVE PRESENT PAST VB-3PS VB-PL) ELIM),

the new feature list will be (VERB TRANSITIVE PRESENT VB-3PS). (ELIM is simply a property indicator chosen to indicate this list which ELIMinates features). Thus, even though words may have more than one part of speech, when they appear in the parsing tree, they will exhibit only those features relevant to their actual use in the sentence.

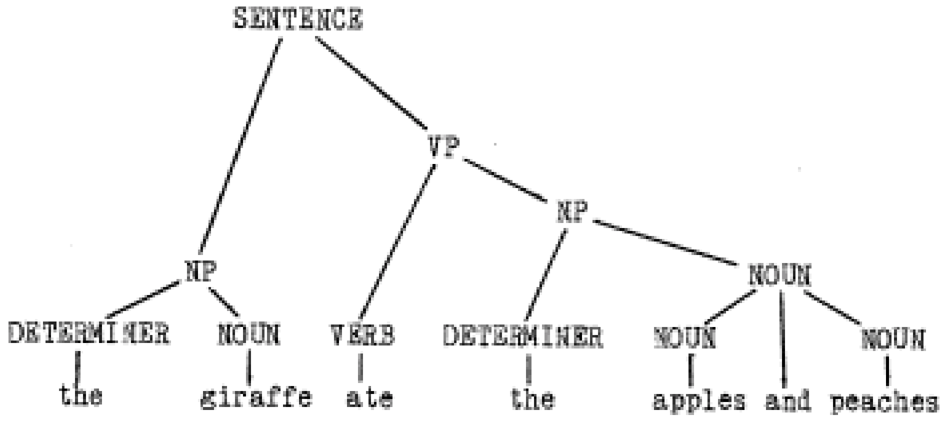
III.2 Special Words

Some words must be handled in a very special way in the grqmmar. The most prevalent are conjunctions, such as "and" and "but". When one of these is encountered, a program should be called to decide what steps should be taken in the parsing. This is done by giving these words the grammatical features SPEC or SPECL. Whenever the function PARSE is evaluated, before returning it checks the next word in the sentence to see if it has the feature SPEC. If so, the SPEC property on the property list of that word indicates a function to be evaluated before parsing continues. This program can in turn call PROGRAMMAR programs and make an arbitrary number of changes to the parsing tree before returning control to the normal parsing procedure. SPECL has the same effect, but is checked for when the function PARSE is called, rather than before it returns. Various other special variables and functions allow these programs to control the course of the parsing process after they have been evaluated. By using these special words, it is possible to write amazingly simple and efficient programs for some of the aspects of grammar which cause the greatest difficulty. This is possible because the general form of the grammar is a program.

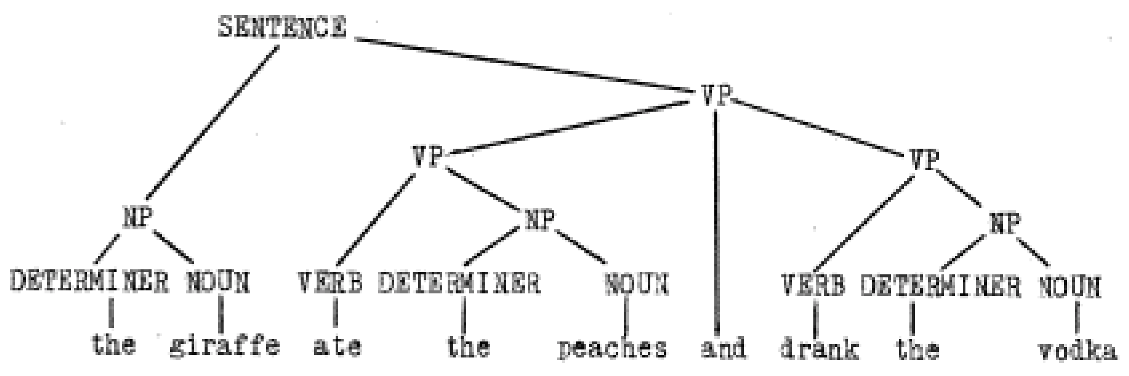
For example, "and" can be defined as a program which is diagrammed:

$$\begin{tikzcd}
\text{Parse a unit of the same type as the currently active node} \arrow[r] \arrow[d] & \text{Return failure} \\
\text{Replace the node with a new node combining the old one and the one you have just found} \arrow[d] & \\
\text{Return success} &
\end{tikzcd}$$

For example, given the sentence "The giraffe ate the apples and peaches." the program would first encounter "and" after parsing the NOUN apples. It would then try to parse a second NOUN, and would succeed, resulting in the structure:



If we had the sentence, "The giraffe ate the peaches and drank the vodka." the parser would first try the same thing. However, "drank" is not a NOUN, so the AND program would fail and the NOUN "apples" would be returned unchanged. This would cause the NP "the apples" to succeed, so the AND program would be called again. It would fail to find a NP beginning with "drank", so the NP "the apples" would be returned, causing the VP to succeed. This time, AND would try to parse a VP and would find "drank the vodka". It would therefore make up a combined VP and cause the entire SENTENCE to be completed with the structure:



The program to actually do this would take only 3 or 4 lines in a PROGRAMMAR grammar. In the actual system, it is more complex as it handles lists (like "A, B, and C") other conjunctions (such as "but") and special constructions (such as "both A and B").

III.3 The Dictionary

Since PROGRAMMAR is embedded in LISP, the facilities of LISP for handling atom names are used directly. To define a word, a list of grammatical features is put on its property list under the indicator WORD, and a semantic definition under the indicator SMNTC. Two facilities are included to avoid having to repeat information for different forms of the same word. First, there is an alternate way of defining words, by using the property indicator WORD1. This indicates that the word given is an inflected form, and its properties are a modified form of the properties of its root. A WORD1 definition has three elements, the root word, the list of features to be added, and the list of features to be removed. For example, we might define the word "go" by: (DEFPROP GO (VERB INTRANSITIVE MOTION INFINITIVE) WORD) We could then define "went" as (DEFPROP WENT (GO (PAST)(INFINITIVE)) WORD1) This indicates that the feature INFINITIVE is to be replaced by the feature PAST, but the rest (including the semantic definition) is to remain the same as for "go".

The other facility is an automatic system which checks for simple modifications, such as plurals, "-ing," forms, "-er" and "-est" forms and so forth. If the word as typed in is not defined, the program looks at the way it is spelled, tries to remove its ending (taking into account rules such as changing "running" to "run", but "buzzing" to "buzz"). It then tries to find a definition for the reduced root word, and if it succeeds, it makes the appropriate changes for the ending (such as changing the feature SINGULAR to PLURAL). The program which does this is the one part of the PROGRAMMAR system described here which is specifically built for English. Everything else described is designed generally for the parsing of any language. In any particular language, this input funtion would have to be written according to the special rules of morphographemic structure. The only requirement for such a program is that its output must be a list, each member of which corresponds to a word in the original sentence, and is in the form described in section III.5. This list is bound to the variable SENT, and is the way in which PROGRAMMAR sees its input.

13

III.4 Backup Facilities

As explained in section II.6, there is no automatic backup, but there are a number of special functions which can be used in writing grammars. The simplest, (POPTO X) simply removes nodes from the tree. The argument is a list of features, and the effect is to remove daughters of the currently active node, beginning with the rightmost and working leftword until one is reached with all of those features. (POP X) is the same, except that it also removes the node with the indicated features. If no such node exists, neither function takes any action. (POP) is the same as (POP NIL), and a non-nil value is returned by both functions if any action has been taken.

A very important feature is the CUT variable. One way to do backup is to first try to find the longest possible constituent at any point, then if for any reason an impasse is reached, to return and try again , limiting the consituent from going as far along in the sentence. For example, in the sentence "Was the typewriter sitting on the cake?", the parser will first find the auxilliary verb "was", then try to parse the subject. It will find the noun group "the typewriter sitting on the cake", which in another context might well be the subject ("the typewriter sitting on the cake is broken."). It then tries to find the verb, and discovers none of the sentence is left. To back up, it must change the subject. A very clever program would look at the structure of the noun group and would realize that the modifying clause "sitting on the cake" must be dropped. A more simple-minded but still effective approach would use the following instructions:

```
(** N PW)
(POP)
((CUT PTW)SUBJECT (ERROR))
```

The first command sets the pointer PTW to the last word in the constituent (in this case, "cake"). The next removes that constituent. The third sets a special pointer, CUT to that location, then sends the program back to the point where it was looking for a subject. It would now try to find a subject again, but would not be allowed to go as far as the word "cake". It might now find "the typerwriter sitting," an analog to "The man sitting is my uncle." If there were a good semantic program, it would realize that the verb "sit" cannot be used with an inanimate object without a location specified. This would prevent the constituent "the typewriter sitting" from ever being parsed. Even if this does not happen, the program would fail to find a verb when it looked at the remaining sentence, "on the cake." By going through the cutting loop again, it would find the proper subject, "the typewriter," and would continue through the sentence.

Once a CUT point has been set for any active node, no descendant of that node can extend beyond that point until the CUT is moved. Whenever a PROGRAMMAR program is called, the variable END is set to the current CUT point of the node which called it. The CUT point for each constituent is initially set to its END. When the function PARSE is called for a word, it first checks to see if the current CUT has been reached, and if so it fails. The third branch in a three-direction branch statement is taken if the current CUT point has been reached. The CUT pointer is set with the function CUT of one argument,.

III.5 Messages

To write good parsing programs, we may at times want to know why a particular PROGRAMMAR program failed, or why a certain pointer command could not be carried out. In order to facilitate this, two message variables are kept at the top level of the system, MES, and MESP. Messages can be put on MES in two ways, either by using the special failure directions in the branch statements (see section II.6) or by using the functions M and MQ, which are exactly like F and FQ, except they put the indicated feature onto the message list ME for that unit. When a unit returns either failure or success, MES is bound to the current value of ME, so the calling program can receive an arbitrary list of messages for whatever purpose it may want them. MESP always contains the last failure message received from ** or *.

III.6 The form of the Parsing Tree

Each node is actually a list structure with the following information:

FE	the list of features associated with the node
NB	the place in the sentence where the constituent begins
N	the place immediately after the constituent
H	the subtree below that node (actually a list of its daughters in reverse order, so that H points to the last constituent parsed)
SM	a space reserved for semantic information

These can be used in two ways. If evaluated as variables, they will always return the designated information for the currently active node. C is always a pointer to that node. If used as functions of one argument, they gi the appropriate values for the node pointed to by that argument; so (NB H) gives the location in the sentence of the first word of the last constituent parsed, while (FE(NB H)) would give the feature list of that word.

Each word in the sentence is actually a list structure containing the 4 items:

FE	as above
SMWORD	the semantic definition of the word (see section III.5)
WORD	the word itself (a pointer to an atom)
ROOT	the root of the word (e.g. "run" if the word is "running").

III.7 Variables Maintained by the System

There are two types of variables, those bound at the top level, and those which are rebound every time a PROGRAMMAR program is called.

Variables bound at the top level

N	Always points to next word in the sentence to be parsed
SENT	Always points to the entire sentence
PT PTW	Tree and sentence pointers. See Section III.6
MES MESP	List of messages passed up from lower levels. See Section III.9

Special variables bound at each level

C FE NB SM H	See section III.2
NN CUT END	See section III.8. NN always equals (NOT(EQ CUT END))
UNIT	the name of the currently active PROGRAMMAR program
REST	the list of arguments for the call to PARSE (These form the inital feature list for the node, but as other features are added, REST continues to hold only the original ones.)
T1 T2 T3	Three temporary PROG variables for use by the PROGRAMMAR program in any way needed.
MVB	Bound only when a CLAUSE is parsed used as a pointer to the main verb
ME	List of messages to be passed up to next level See Section III.9

III.8 Pointers

The system always maintains two pointers, PT to a place on the parsing tree, and PTW to a place in the sentence. These are moved by the functions * and ** respectively, as explained in section II.7. The instructions for PT are:

C	set PT to the currently active node
H	set PT to the most recent (rightmost) daughter of C
DL	(down-last) move PT to the rightmost daughter of its current value
DLC	(down-last completed) like DL, except it only moves to nodes which are not on the push-down list of active nodes.
DF	(down-first) like DL, except the leftmost
PV	(previous) move PT to its left-adjacent sister
NX	(next) move PT to its right-adjacent sister
U	(up) move PT to the parent node of its current value
N	Move PT to the next word in the sentence to be parsed

The pointer PTW always points to a place in the sentence. It is moved by the function ** which has the same syntax as *, and the commands:

N	Set PTW to the next word in the sentence to be parsed
FW	(first-word) set PTW to the first word of the consstituent pointed to by PT
LW	(last-word) like FW
AW	(after-word) like FW, but first word after the constituent

NW	(next-word) Set PTW to the next word after its current value
PW	(previous-word) like NW
SFW	(sentence-first-word) set PTW to the first word in the sentence
SLW	(sentence-last-word) like SFW

Since the pointers are bound at the top level, a program which calls others which move the pointers may want to preserve their location. PTW is a simple variable, and can be saved with a SETQ, but PT operates by keeping track of the way it has been moved, in order to be able to retrace its steps. This is necessary since LISP lists are threaded in only one direction (in this case, from the parent node to its daughters, and from a right sister to its left sister). The return path is bound to the variable PTR, and the command (PTSV X) saves the values of both PT and PTR under the variable X, while (PTRS X) restores both values.

III.9 Feature Manipulating

As explained in section II.7, we must be able to attach features to nodes in the tree. The functions F, FQ, and TRNSF are used for putting features onto the current node, while R and RQ remove them. (F A) sets the feature list FE to the union of its current value with the list of features A. (FQ A) adds the single feature A (i.e. it quotes its argument). (TRNSF A B) was explained in Section II.7. R and RQ are inverses of F and FQ. The functions IS, ISQ, CQ, and NQ are used to examine features. If A points to a node of the tree or word of the sentence, and B points to a feature, (IS A B) returns non-nil if that nodYhas that feature. (ISQ A B) is equivalent to (IS A (QUOTE B)), (CQ B) is the same as (ISQ C B) (where C always points to the currently active node), and (NQ B) is the same as (ISQ N B)(N always points to the next word in the sentence left to be parsed).

IV Examples of Sentences Parsed

This section demonstrates the use of PROGRAMMAR on two English sentences. They were parsed using the current English grammar for the system. In the examples, a number of features are used. It is difficult to explain their significance without a thorough explanation of the systems from which the features were selected. This grammar will be fully explained in a forthcoming paper.

The form of input to PROGRAMMAR is the sentence in normal orthography and punctuation. The actual console dialogue is reproduced, with the inputs underlined. The function WALLP is one of a number of functions used to examine the parsing tree from the console. It prints out a representation of the tree in a readable form. In normal use, after finishing the parsing, PROGRAMMAR would continue with semantic analysis. For use without semantics, it halts to allow the user to examine the results.

```
(PROGRAMMAR)

READY/

 HOW MANY EGGS WOULD YOU HAVE BEEN GOING TO USE IN THE CAKE IF YOU HADN'T
 LEARNED YOUR MOTHER'S RECIPE WAS WRONG?

*OK
LISTENING

(WALLP C)
```

```
(((HOW MANY EGGS WOULD YOU HAVE BEEN GOING TO USE IN THE
      CAKE IF YOU HADN'T LEARNED YOUR MOTHER'S RECIPE WAS WRONG)
  (CLAUSE MAJOR QUEST NGQUES POLR2 ACTV OBJ1Q TRANS)

       (((HOW MANY EGGS) (NG QUEST HOWMANY NDEF NPL DET)

                    ((HOW (QDET))
                    (MANY (QDET))
                    (EGGS (H NPL))))

        (WOULD (VB AUX MODAL QAUX))
        ((YOU) (NG SUBJ DEF NS NPL)  ((YOU (PRON NPL NS SUBJ OBJ))))
        ((HAVE BEEN GOING TO USE) (VG MODAL NAGR (FUT PAST MODAL))

                           ((WOULD (VB AUX MODAL QAUX))
                            (HAVE (HAVE VB AUX VO TRANS))
                            (BEEN (AUX VB BE EN))
                            (GOING (VB ITRNS ING))
                            (TO (TO))
                            (USEP(VB VO TRANS MVB))))

        ((IN THE CAKE) (PREPG)

                    ((IN (PLACE PREP PLACE))
                    ((THE CAKE)(NG OBJ DET NS DEF)

                              ((THE (DET NPL NS DEF))
                              (CAKE (H NS))))))

        ((IF YOU HADN'T LEARNED YOUR MOTHER'S RECIPE WAS WRONG)
         (CLAUSE BOUND DECLAR ACTV TRANS)

              ((IF (BINDER))
              ((YOU) (NG SUBJ DEF NS NPL)  ((YOU (PRON NPL NS SUBJ OBJ))))
              ((HADN'T LEARNED) (VG VPL V3PS NEG (PAST PAST))

                              ((HADN'T (HAVE VB AUX TRANS PAST VPL V3PS VFS NEG))
                               (LEARNED (VB TRANS REPOB PAST EN MVB))))

              ((YOUR MOTHER'S RECIPE WAS WRONG)
               (CLAUSE RSNG REPORT OBJ OBJ1 DECLAR BE INT)

                  (((YOUR MOTHER'S RECIPE)
                     (NG SUBJ NS DEF DET POSES)

                     (((YOUR MOTHER'S)
                       (NG SUBJ NS DEF DET POSES POSS)

                       (((YOUR) (NG SUBJ POSS)
                          ((YOUR (PRON NPL NS SUBJ OBJ POSS))))
                        (MOTHER'S (H NS POSS))))
```

```
                         (RECIPE (H NS))))

                      ((WAS) (VG V3PS VFS (PAST))
                         ((WAS (AUX VB BE V3PS VFS PAST MVB))))
                      ((WRONG) (ADJG Q COMP)  ((WRONG (EP))))))))))
           )
```

READY/

<u>PICK UP ANYTHING GREEN, AT LEAST THREE OF THE BLOCKS, AND EITHER A BOX OR A SPHERE WHICH IS BIGGER THAN ANY BRICK ON THE TABLE.</u>

```
*OK
LISTENING
```

<u>(WALLP C)</u>

```
(((PICK UP ANYTHING GREEN /, AT LEAST THREE OF THE BLOCKS /, AND EITHER
    A BOX OR A SPHERE WHICH IS BIGGER THAN ANY BRICK ON THE TABLE)
  (CLAUSE MAJOR IMPER ACTV TRANS)

  (((PICK) (VG IMPER)  ((PICK (VPRT VB VO TRANS MVB))T)
   (UP (PRT))
   ((ANYTHING GREEN /, AT LEAST THREE OF THE BLOCKS /, AND EITHER
     A BOX OR A SPHERE WHICH IS BIGGER THAN ANY BRICK ON THE TABLE)
    (NG OBJ OBJ1 EITHER COMPOUND LIST NS)

    (((ANYTHING GREEN) (NG OBJ OBJ1 TPRON)

       ((ANYTHING (NS TPRON))
        (GREEN (EP))))

     ((AT LEAST THREE OF THE BLOCKS)
      (NG OBJ OBJ1 COMPONENT NUMD NUM NPL DET OF)

          ((AT (AT))
           (LEAST (NUMD NUMDAT))
           (THREE (NUM))
           ((OF THE BLOCKS)
             (PREPG OF)

                     ((OF (PREP))
                      ((THE BLOCKS)
                       (NG OBJ DET NPL DEF)

                                  ((THE (DET NPL NS DEF))
                                   (BLOCKS (H NPL))))))))

     ((A BOX OR A SPHERE WHICH IS BIGGER THAN ANY BRICK ON THE TABLE)
      (NG OBJ OBJ1 COMPONENT OR COMPOUND BOTH NS)

      (((A BOX)
        (NG OBJ OBJ1 COMPONENT DET NS INDEF)
     ((A (DET NS INDEF))
      (BOX (H NS))))

((A SPHERE WHICH IS BIGGER THAN ANY BRICK ON THE TABLE)
 (NG OBJ OBJ1 COMPONENT DET NS INDEF)

 ((A (DET NS INDEF))
  (SPHERE (H NS))
  ((WHICH IS BIGGER THAN ANY BRICK ON THE TABLE)
   (CLAUSE RSQ SUBREL BE INT)

   (((WHICH) (NG RELWD DEF NPL)  ((WHICH (NPL))))
    ((IS) (VG V3PS (PRES))  ((IS (AUX VB BE V3PS PRES MVB))))
    ((BIGGER THAN ANY BRICK ON THE TABLE)
     (ADJG Q COMP COMPAR THAN)

     ((BIGGER (EP COMPAR))
      (THAN (THAN))
      ((ANY BRICK ON THE TABLE)
       (NG SUBJ COMPAR DET NS QNTFR)

       ((ANY (DET NS NPL QNTFR))
        (BRICK (H NS))
        ((ON THE TABLE)
         (PREPG Q)

         ((ON (PREP PLACE))
          ((THE TABLE)
           (NG OBJ DET NS DEF)

           ((THE (DET NPL NS DEF)) (TABLE (H NS))))))))))))))))))))
```

References

1) Daniel Bobrow, "Syntactic Theory in Computer Implementations" in Borko, ed., AUTOMATED LANGUAGE PROCESSING, Wiley 1967

2) M.A.K. Halliday, "Categories of the Theory of Grammar," WORD 17, 1961

3) ______, "Some Notes on 'Deep' Grammar," JOURNAL OF LINGUISTICS 2, 1966

4) ______, "Notes on Transitivity and Theme in English, JOURNAL OF LINGUISTICS 3, 1967

5) Terry Winograd, "Linguistics and the Computer Analysis of Tonal Harmony," JOURNAL OF MUSIC THEORY 12, 1968

6) ______, "An Interpretive Theory of Language, unpublished term paper, 1968

Borko p
1947