MEMORANDUM
RM-6265/1-PR
APRIL 1970

THE MIND SYSTEM: A GRAMMAR-RULE LANGUAGE

Ronald M. Kaplan

PREPARED FOR:
UNITED STATES AIR FORCE PROJECT RAND

The RAND Corporation
SANTA MONICA • CALIFORNIA

MEMORANDUM
RM-6265/1-PR
APRIL 1970

THE MIND SYSTEM: A GRAMMAR-RULE LANGUAGE

Ronald M. Kaplan

This research is supported by the United States Air Force under Project RAND—Contract No. F44620-67-C-0045—monitored by the Directorate of Operational Requirements and Development Plans, Deputy Chief of Staff, Research and Development, Hq USAF. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of Rand or of the United States Air Force.

The RAND Corporation
1700 MAIN ST. • SANTA MONICA • CALIFORNIA • 90406

Published by The RAND Corporation

PREFACE

This is one of a series of papers describing the design, implementation, and use of the MIND* system. The design goals for the MIND system are responsive to the increasingly urgent need for a means of fast and accurate information transactions between relatively senior command, control, and policymaking personnel, on the one hand, and very large, heterogeneous, loosely—formatted information banks on the other. The system is an unobtrusive servant; it understands, acts upon, and replies with, English sentences; its users will require no special training. It is thus a prototype for a class of systems that are well suited to the critical task of unifying, controlling, and exploiting the massive and often chaotic flow of information that centers upon the senior command levels of the Air Force and other services, especially in emergency situations.

The MIND system consists of nested and chained modules of high level programming language statements, and it is therefore relatively easy to modify, either for improvement or for adaptation to specialized applications.

*Management of Information through Natural Discourse.

SUMMARY

This paper gives a formal description of the MIND Grammar–Rule Language, a notation for the syntactic rules of natural languages. The notation differs from others that have been proposed in that (1) it is completely specified by means of a formal grammar in the manner that has become standard with programming languages, (2) it embodies a grammatical formalism designed for application to automatic grammatical analysis rather than the generation of sentences or the enshrinement of a universal theory of human language, (3) it is based on unrestricted rewriting rules operating on strings of trees rather than phrase–structure or transformational rules (though these are both allowed for), (4) it is intended to uncover the deep structures of sentences, (5) it is an exceedingly rich notation allowing a great many generalizations to be captured and allowing a single rule to stand for large numbers of rules in a simpler system, (6) the objects referred to in the rules are characterized by unrestricted lists of features each of which takes the form of an attribute and an associated value so that there is no limit in principle to the refinements that can be introduced into a grammar. The paper also contains a sketch of the strategy to be used in the computer for applying the rules and some examples of rules that might occur in grammars of English.

CONTENTS

THE MIND SYSTEM: A GRAMMAR-RULE LANGUAGE

1. INTRODUCTION

This paper precisely defines the MIND* Grammar-Rule Language (MGRL). MGRL was developed as the formalism for the grammar input to the MIND syntactic analysis component, which is being programmed for the IBM 360/65 at RAND. Hence the semantic interpretation of the various elements will be given in terms of the effect they will have on the parsing process. The major criteria underlying the design of this language are that it should (1) facilitate the specification of enough information to make the parsing procedure operate efficiently, and (2) allow significant linguistic generalizations to be stated perspicuously. At the same time, however, it should (3) be flexible and powerful enough so that it makes a minimal number of presuppositions about the nature of syntactic generalizations. Concomitant with this flexibility and power is the capability of denoting a single parsing action in several ways. MGRL is constructed so that (4) the simplest notation for expressing an action will result in the most efficient computational execution of that action.

This paper is divided into several parts. A general impression of MGRL can be obtained from Sections 2 and 5. Section 2 gives an overview of the basic parsing strategy,

*Management of Information through Natural Discourse

while Section 5 provides examples of realistic English grammar rules written in MGRL. These are compared with corresponding rules specified in more conventional linguistic notations. The detailed definition of MGRL is presented in Section 4; the metalinguistic conventions used in the formal description are set forth in Section 3.

2. GENERAL DESCRIPTION OF THE PARSER

The input to the parser is a family of strings of trees. The trees closely resemble those of ordinary linguistic descriptions, except that the node labels consist of lists of properties expressed as attribute-value pairs. These property lists resemble "complex symbols" (Chomsky, 1965), but they label all nodes, not just lexical categories, and the attributes are not necessarily binary-valued. A tree may be degenerate in the sense that it only has one node. The initial family of strings arises in the following way: each word of the original text is looked up in a dictionary by a program (the morphological analyzer) which replaces it by a set of sequences of trees. In the simplest case an original word will give rise to a set whose elements are single trees. A proper noun might be represented by a tree with a root labeled "part-of-speech: noun-phrase" dominating a determiner and a noun. (Proper nouns do not take determiners in English, but if one is supplied by the dictionary, all noun phrases have the same form and the number of special cases the parser must recognize is reduced.) Other words might result in sets whose elements are sequences of two trees. For example, corresponding to a verb there might be a sequence of two trees, the first of which represents the tense, number, and person of the verb, and the second of which identifies the verb itself. This is a reasonable arrangement because,

in English at least, tense, number, and person morphemes appear sometimes with the main verb and sometimes with an auxiliary. If they are represented by a separate member of the sequence, it is easy to move them from one position to the other, and many of the same parsing rules can therefore be used to analyze sets of superficially different sentences, such as

(1) John wrote a novel.

(2) Did John write a novel?

(3) Was John writing a novel?

If the first member of a verb sequence is a degenerate tree representing tense, number, and person, the second tree might consist of a head node labeled "part—of—speech: verb" dominating a node for the verb itself, perhaps a node indicating what kind of subject it takes, a node showing what kind of object, if any, and so on.

For each word the morphological analyzer provides a <u>set</u> of sequences of trees because words are, in general, syntactically or semantically ambiguous. The dictionary entry for the word "like" must have one sequence for its verbal interpretation and other sequences for its interpretations as noun, adjective, and preposition. Consider the sentence

(4) "He has many likes and dislikes."

Before the parser has done its work, there is no way to tell if "likes" is to be processed as a noun or verb, and

there will therefore be at least two sequences in the set for this word. The morphological analyzer, however, can determine that it is not an adjective or a preposition because these cannot occur with the suffix "s". In general, the set of sequences provided for a word by the morphological analyzer will be a subset of the set of sequences contained in the dictionary entry of the word. The word "has" will probably have two sequences in its set, one for its use as an auxiliary and one for its use as a main verb. "Dislikes" will also have two sequences because it has the same grammatical properties as "likes". The words "he", "many", and "and" presumably have one interpretation each, and their sets are therefore singletons.

A string in the initial family consists of the concatenation of sequences of trees, one sequence from the set of each word of the original sentence, in the order in which the words appear in the sentence. An initial family contains one such string for each of the possible combinations of sequences taken from the various sets. Thus the number of strings in an initial family is the product of the numbers of sequences in the sets of the individual words. For the sentence (4), this number is eight. If the sentence is grammatically unambiguous, only one of these strings will lead to a successful parse, if the parsing grammar is adequate. If the sentence is in fact ambiguous, then more than one member of the initial

family of strings of trees will result in a successful parse.

Conceptually the effect of each grammar rule applied by the parser is to examine the family of strings and, for each string satisfying conditions specified by the rule, to add to the family a new string whose structure is determined conjointly by the string in question and the rule. Thus the family of strings representing the sentence is constantly growing and changing; at any stage in the parsing process the current family of strings is called the chart. A substring of a string in the chart is a chart section, and the conditions given in the rules describe the structure and properties of chart sections. A string in the chart satisfies the conditions of a rule if and only if it contains a substring meeting the appropriate description. A chart section meeting the description given in a rule is said to match the rule, and the new string added to the chart is the same as the old one except that the matching chart section is replaced by a new substring. Notice that (1) the strings in the chart are made up of trees even though they may be degenerate and even though they may often be identified only by properties associated with their top nodes; (2) that a rule identifies a string to which it applies by properties of a certain chart section that it contains, and this is precisely the substring that is replaced in order to arrive at the new

string to be added to the chart; and (3) that, with a notable exception to be discussed shortly, the action of a rule does not cause the removal of any string from the current family.

This is the effect of a grammar rule conceptually. What happens inside the computer is in fact different and considerably more complicated because the straightforward strategy would entail an unthinkable expenditure of space and time. One difference is particularly important to the understanding of the rule language: the explication above centers on strings of trees—conceptually these are the objects which are examined by the parser and which are added to the chart. But the rules specify conditions on substrings, and strings added to the chart differ from strings already in the chart only within the matching chart section. Thus, in order to increase speed and efficiency, the computer actually operates only with substrings and never considers whole strings. It only examines chart sections, and it only adds new substrings to create an alternative in the original string to the matching chart section. This being the case, the rule language and the description of the rule language deal almost exclusively with chart sections rather than complete strings. Conceptually or computationally, the final result is the same: when no more rules can be applied, then every string in the chart consisting of a single tree whose top

node is labeled "part-of-speech:sentence" (cf. p.19) is a successful parse of the sentence.

Normally, a rule application does not cause the removal of a string from the current family. However, it is possible to designate a rule as deterministic, in which case it acts like other so-called non-deterministic rules except that, when the new string is added to the chart, the old one is deleted. Suppose that the string A B C belongs to the chart and that the grammar contains a pair of rules which, for the present, can be written as follows:

(5) a b = d

(6) b c = e

A, B, C, D, and E are complete trees and a, b, c, d, and e are specifications of properties that trees must have for (5) and (6) to apply to them. Suppose that A, B, C, D, and E have the properties specified by a, b, c, d, and e, respectively. Both rules will apply and two new strings, namely D C and A E, are added to the chart. However, if either of the rules is deterministic, their effect cannot be predicted without knowing the order in which the parser will attempt to apply them. If a deterministic rule is applied first, it may, and in this case it will, destroy a string to which another rule would otherwise have applied. The basic strategy of the parser is to search each string of the chart from right to left for substrings that meet the requirements of some rule in the grammar and, for each position at which such a substring might start, to attempt

to apply the rules in the order in which they appear in the grammar. Thus, in this example, the deterministic rules will actually replace A B C by A E only, since this results from operating on a substring further to the right in the original string. However, in attempting to apply an individual rule to a given chart section, the parser processes both in the opposite order, strictly from left to right.

Finally, a word should be said about the relationship of MGRL to other linguistic formalisms. It is easy to show that, except for implementation limitations, for every unrestricted rewriting grammar (Turing machine), there is an equivalent MGRL grammar. The rules of the equivalent MGRL grammar will contain only strings of degenerate trees. The property lists of the root nodes (the only nodes) of these trees will be in one to one correspondence with the vocabulary items of the rewriting grammar. Likewise, every conventional transformational grammar has an MGRL equivalent. In this case, the MGRL rules will contain only degenerate strings of trees, a single tree corresponding to each structural description occurring in a transformational rule. A specification of a list of MGRL transformations will correspond to each structural change. Thus it is clear that the MGRL formalism has at least the generative power of Turing machines and of transformational grammars.

3. THE METALANGUAGE

This section describes the metalinguistic conventions which are used in defining MGRL. For the most part, these conventions are adapted from other linguistic formalisms.

3.1 Notation

MGRL is basically a context-free phrase-structure language and is thus defined by phrase structure rewrite rules with a single metalinguistic variable on the left side and its immediate constituents on the right. The symbol → separates the left and right sides, and may be read as "is rewritten as" or "has as constituents". Metalinguistic variables are written in underlined lower-case letters and are separated from each other by spaces. Thus $\underline{xyz} \rightarrow \underline{a}\ \underline{b}\ \underline{c}$ is a well-formed meta-rule.

The other symbols used in the metalanguage are square brackets, a superscript "n", and braces. Brackets indicate that the sequence of symbols enclosed within them are optional; they are also used to enclose and group the arguments of macro definitions (cf. Section 3.2). The superscript indicates that the immediately preceding symbol may occur n times in a sequence, for $n \geq 1$. The asterisk and brackets can be used together to indicate that an element may occur zero, one, or more times. The convention has been adopted that brackets act as grouping

symbols in addition to their normal function when succeeded by a superscript. Thus $[\underline{a}\ \underline{b}\ \underline{c}]^n$ means that the sequence $\underline{a}\ \underline{b}\ \underline{c}$ may occur an optional number of times. Finally, the braces indicate that a choice must be made between two or more alternatives written on different lines enclosed within the braces. Hence, $\left\{\begin{matrix}\underline{a}\\ \underline{b}\end{matrix}\right\}$ means that either $\underline{a}$ will do or $\underline{b}$ will do, but not both at the same time.

All other symbols which appear in the formal description are object-language symbols, that is, they actually occur within grammar rules.

3.2 Macro Definitions

The metalanguage includes a macro definition facility which is used to express generalizations about MGRL that are not easily stated with simple phrase-structure rules. When a sequence of meta-rules appears which differs from other sequences only in minor details, then the similarities can be captured in a macro definition while the variations can be designated by particular values of the macro arguments. A macro definition is invoked by the appearance of the metalanguage name of the macro at the point where one of the rule sequences would otherwise have occurred. The name of the macro is followed by particular argument values enclosed in square brackets and separated by commas. The invocation of a macro is understood in the following way: the meta-rule in which the macro name appears is interpreted as if the name and

its arguments were replaced by the sequence of symbols to the right of the double arrow in the definition of the macro. The double arrow signifies this syntactic substitution procedure. In making this substitution, the values of the arguments given in the invocation replace all occurrences of the corresponding formal arguments in the expansion of the macro, and in the phrase-structure rules included in the macro definition. The name of a macro consists of uppercase letters preceded by a percent-sign. The formal parameters are in uppercase and are underlined.

Suppose %MAC is a macro invoked by the meta-rule (7):

(7) alpha → beta gamma %MAC[a,b,/]

If %MAC is defined by (8)-(9), then (7) should be

(8) %MAC[X,ARG,SLASH] ⇒ Xdelta

(9) Xdelta → Xepsilon (ARG SLASH [beta])

interpreted as if it were (10), and (11) should be included as a meta-rule:

(10) alpha → beta gamma adelta

(11) adelta → aepsilon (b / [beta])

It will often be the case that a macro argument in an invocation is the special symbol Ø. This signifies that all occurrences of the corresponding formal argument are to be replaced by the null symbol in the expansion of the macro, that is, they are to be deleted. Thus the expansion of %MAC[Ø,b,Ø] would result in (12) instead of (11):

(12) delta → epsilon (b [beta])

The metalanguage contains two macros, %BOOL and %SET. At several places in the object language, Boolean expressions of some basic element are allowed, and the macro %BOOL has been constructed to capture this fact clearly and simply:

(M1) %BOOL[X,AND,NEG] ⇒ Xalternation

(M2) Xalternation → Xconjunction [| Xconjunction]^{n}

(M3) Xconjunction → Xitem [AND Xitem]^{n}

(M4) Xitem → [NEG] $\left\{ \begin{matrix} X \\ (\text{Xalternation}) \end{matrix} \right\}$

This sequence of statements will generate a Boolean expression of elements of type X, where | is the or-operator, AND is the and-operator, and NEG is the negation-operator. The particular realizations of X, AND, and NEG vary from invocation to invocation, and for some, AND and NEG are replaced by ∅.

%SET defines object language constituents which are interpreted, conceptually and procedurally, as ordinary sets.

(M5) %SET[X,PREFIX,MINUS,VARMOD] ⇒ Xset

(M6) Xset → Xsingleton [, Xsingleton]^{n}

(M7) Xsingleton → [PREFIX] [MINUS] $\left\{ \begin{matrix} \text{literal} \\ [\#]\ \text{valvar} \\ (\text{Xset}) \end{matrix} \right\}$ [VARMOD^{n}]

A set is essentially a sequence of singletons separated by commas. The commas signify disjunction or union as in ordinary set notation. An elementary member of a set is a literal, and this is the central constituent of Xsingleton. The valvar alternative is another way of representing literals, while the parenthesized Xset permits the PREFIX, MINUS, and VARMOD options to operate on a sequence of Xsingletons. A parenthesized Xset is interpreted as a subset of the Xset containing the Xsingleton, not as an element of that set. The significance of sets will be discussed in more detail when the macro definition is invoked.

4. THE OBJECT LANGUAGE

This section comprises the main body of the paper. It describes in detail how cards and symbols are physically and logically arranged in a well-formed grammar. The first sub-section describes options which are not logically a part of MGRL and which are ignored by the parser. The remainder of this section falls roughly into two parts: sub-sections 4.2 through 4.7 define the pictorial elements of the rule language, those constituents which explicitly outline strings and subtrees in the chart. Sub-sections 4.8 through 4.10 characterize the non-pictorial rule elements, those which describe the chart and the operations the parser can perform on it more or less implicitly. These latter rule constituents provide much of the power and flexibility of MGRL.

4.1 Comments, Spaces, and Card Location

As far as the computer implementation is concerned, the language has been designed so that spaces and comments may occur anywhere to enhance readability and comprehensibility. Thus the object-language sequence AB C D will have the same effect as the sequence ABCD or the sequence A BCD. Comments are set off on the left by the character sequence /* and on the right, */, just as in PL/1; comments may occur even within a single rule.

The implementation allows the further freedom that the location of rules on cards does not matter. A rule

can begin in any column and can extend to any number of cards. In addition, more than one rule can appear on any given card, if space permits.

4.2 Low-Level Constituents

At the outset the elementary syntactic units out of which the MGRL is built will be described. This will make the examples given in the explication of larger structures easier to understand. The alphabet of the object-language contains the 26 upper-case English letters and the digits 0 through 9. With these the following definitions are made:

(D1) $\underline{\text{character}} \rightarrow \left\{ \begin{matrix} \underline{\text{letter}} \\ \underline{\text{digit}} \end{matrix} \right\}$

(D2) $\underline{\text{variable}} \rightarrow \underline{\text{character}}^{n}$

(D3) $\underline{\text{literal}} \rightarrow \underline{\text{character}}^{n}$

(D4) $\underline{\text{number}} \rightarrow \underline{\text{digit}}\ [\underline{\text{digit}}\ [\underline{\text{digit}}]]$

(D5) $\underline{\text{litlist}} \rightarrow \underline{\text{literal}}\ [,\ \underline{\text{literal}}]^{n}$

(D6) $\underline{\text{declaration}} \rightarrow \underline{\text{number}}\ (\ \underline{\text{litlist}}\ )\ [(\ \underline{\text{litlist}}\ )]$

The following are valid variables or literals:

(13) M

(14) 12

(15) M47A

(16) NOUN

(17) PARTICLE

variables and literals have the same definition, but they are clearly distinguished by the context in which they appear. The same sequence of characters may be used as both a literal and a variable without confusion. At present there are four implementation limitations: first, a maximum of 256 distinct variables are allowed in any single rule. Second, a maximum of only 254 distinct literals are permitted in a complete grammar. Third, variables and literals are both truncated after seven characters, and thus, for example, the parser will not distinguish PARTICIPLE from PARTICIPLES or PARTICIPATE. Fourth, any ordinary one, two, or three digit integer is a number, provided that its value is between 0 and 255. 6, 127, and 0 qualify as number, but 301 does not.

With declarations the user can determine the internal representations of the literals used in the grammar rules and in the lexicon. A declaration starts with a number which, when converted to binary form, is the 8-bit representation of all the literals in the first litlist; this number must be less than 254 since the parser reserves the bit-codes of 254 and 255 for internal processing. The user, by including in the list more than one literal, can specify sets of synonymous literals, synonymous in that the parser will not distinguish between them. This facility might be employed in two situations: first, when the context in the rules clearly separates the synonyms, it

may be used to help compress a large number of literals into the 254 allowable internal forms. Second, if literals are conceptually distinct but the distinction is inconsequential at an early stage of grammar development, they may be given different names and then be mapped into the same internal form by a declaration. When the distinction becomes significant, it can be made throughout the grammar simply by removing the declaration.

The literals in the second litlist are defined in the declaration to be synonyms of each other and antonyms of the literals in the other list. Thus SING might be declared to be the opposite of PLUR, and this would define the result of an "alpha switching" operation. (This is discussed in more detail on page 42.) For any n between 0 and 126, the internal representations 2n and 2n + 1 are opposites, so that 24(A)(B) has the same effect as 25(B)(A) and as the pair of declarations 24(A) and 25(B). Note that "opposite" is a symmetric relation.

The declaration facility has one other important function, namely, to control the order in which the parser stores and searches lists of literals. literals are stored in ascending numerical order in the chart and this is the order in which they are encountered in searching. Consequently, efficiency can be increased without otherwise affecting the parsing procedure by declaring frequently occurring literals to have low numbers. For example, if

PS signifies "part of speech," it would appear quite often and should be given the declaration

(18) 1(PS)

Other examples of declaration:

(19) 25(SING, SG)(PL, PLUR)

(20) 124(PAST, FEM, ADV)

If a literal does not appear in a declaration, then the parser automatically assigns it the lowest unused internal representation when the literal is first encountered in a rule.

Three bit-codes, 0, 1 and 253, have special, implementation-defined interpretations. Any literal with bit-code 0 will be synonymous with every other literal, and in an environment requiring the literal to be interpreted as a set, it denotes the "literal universe," the set containing all 254 possible internal representations. Thus 0(ANY) will indicate that ANY is a "universal" literal. The internal code 1 is privileged in two ways: first, since syntactic rules are often defined in terms of the values of a "syntactic category" or "part-of-speech" attribute, MGRL permits the abbreviation of rules by omission of this attribute (cf. Section 4.7). Whenever an attribute is missing, the parser supplies the internal code 1. Second, the bit-code 1 is used to mark a successful parse: when no more rules can be applied, a chart string consisting of a single tree with a root labeled 1:1 is a parse of the input sentence. PS:SENTENCE would denote a successful parse if the declaration 1(PS,SENTENCE) appeared.

Finally, the attribute with bit-code 253 will always have as its value the terminal symbol of a node. This could be a concept number, a character sequence, or perhaps even a matrix of phonological features: the particular form and interpretation of terminals is left up to the user. However, a single input sentence may not contain more than 256 terminal symbols, and these may be referred to only by the ANY literal, that is, by the pair 253:0, or by variable assignments (cf. Sections 4.6 and 4.8).

4.3 Rules

The metalanguage specifies exactly what a well-formed grammar is, so the starting symbol for the generation of the object language is grammar:

(D7) grammar → [declaration^{n} ;] rule^{n}

(D8) rule → [D] id , rule body ;

(D9) id → number [/ [number] [/ number]]

(D10) rule body → left side = [right side]

A grammar is an optional sequence of declarations followed by a sequence of rules. declarations function as described above. A rule begins with an optional letter D and an identification field. If the D appears, it signifies that the rule is deterministic and will cause the deletion of all matching chart sections. If the D is absent, the rule is applied in the regular, non-deterministic manner.

The first number in the id must be greater than zero; it locates this rule in the ordered sequence of other rules in the grammar. Thus at most 255 rules are allowed in a grammar, and their numerical order corresponds to the order in which they are stored, although they need not be numbered consecutively. Therefore, if two rules have the same number, the second will replace the first. The other two numbers in the id allow the user to specify certain ordering restrictions among the various rules. For example, a 25 in the second field might mean that this rule can be applied to any section of the chart which matches, provided that no element in that section has resulted from the application of a rule with first number greater than 25. Thus, the rule with id 26/25 could not apply to its own output, while the id 20/50 would provide for a cyclic ordering of rules. The particular ordering regulations will be precisely described in another publication. Examples of id:

(21) 27
(22) 14/32
(23) 45//6
(24) 225/13/72

If a number is absent, it is interpreted as a zero and does not restrict the rule ordering in any way. Thus the following are equivalent:

(25) 1
(26) 1/0
(27) 1/0/0
(28) 1//0

The rule body contains the complete description of the rule's operation. The left side represents the pattern which must be matched with the chart. The right side specifies the new sections that are to be added to the chart. Note that a rule is terminated by a semicolon.

(D11) left side → $\left\{ \begin{matrix} \text{lhs} \\ [\text{lhs}]\ "\ \text{lhs}\ "\ [\text{lhs}] \end{matrix} \right\}$

(D12) right side → rhs [" transformation^n]

left side is a propositional function which is assigned a truth-value according to the structure of the current chart section. A match occurs if and only if this value is TRUE. Even though left side and right side have some formal similarities, right side is not interpreted as a propositional function. Both left side and right side also identify and tag constituents of the chart for later reference within the same rule.

The pattern described by the left side can be divided into three parts by double-quotes. Anything that appears before the first quote or after the second is considered to be context. For the left side to match, the context and the lhs within the quotes must both match. However, additions to the chart will be made only as alternatives to the chart section that matches the central lhs, resulting in increased parsing efficiency. If every part of the pattern is susceptible of being changed, then no context is given and the double quotes may be omitted. Examples of left side:

(29) <A:B> <C:D> " <Z:W> " <E:F>

(30) "<PS:NOUN>"

(31) <PS:NOUN>

(32) "<X:Y>" <E:F>

In (29) both left and right context are given. (30) and (31) have no context and are equivalent to each other. (32) has only right context.

If the match is successful, the right side denotes the changes to be made. The right side is realized as an rhs followed by an optional sequence of transformations. The rhs describes the section which is to be added to the chart and, in a deterministic rule, is to replace the section of chart that matched the non—context lhs. rhs is somewhat more constrained than lhs in notation and interpretation, since the chart section to be constructed must be precisely and unambiguously specified. The transformations provide another way of specifying the changes which are to be made. A transformation operates on the section of the chart constructed in accordance with the rhs of the rule; if no rhs appears, the transformations are disregarded. transformations will be discussed in more detail in Section 4.10.

If the rule body does not have a right side, the rule has the effect of deleting (deterministically or non—deterministically, depending on the presence or absence of an initial D) the non—context lhs of the left side. Examples of right side:

(33) <A:B_C:D>.X <E:F> "RSA('X,<G:H>)

(34) <Q:R> | <S:T>

(35) <B:C>.1

(36) <U:V>.1 "LCA(1,'NP)

4.4 Lhs and Rhs

(D13) lhs → %BOOL[lterm,Ø,¬]

(D14) rhs → %BOOL[rterm,Ø,Ø]

An lhs is a Boolean combination of lterms with the and-operator denoted by Ø and negation by the symbol ¬. When two lterms are concatenated, they can only match a section of the chart which has one element matching each of the lterms. Furthermore, the elements in the chart must be ordered in the same way as the corresponding lterms in the rule. In deciding whether a section of the chart matches an lhs, the parser assigns a truth-value to each lterm in the Boolean expression: this value is TRUE if the lterm matches the appropriate chart elements and FALSE otherwise. The whole lhs is then evaluated to TRUE or FALSE according to the rules of ordinary propositional calculus. If it is TRUE, then the section of the chart matches the whole lhs. Examples of lhs:

(37) <PS:DET> <PS:ADJ> <PS:VERB>

(38) <PS:VERB> | (<AUX:PLUS> <PS:PART>).VB

(39) <PS:ADV> | <PS:PREP> ¬ <PTYP:NOUN>

rhs has almost the same Boolean form as lhs, the lack of the negation operator being the only formal difference. Remember, however, that rhs is not a propositional function but rather is a description of a substring of trees to be added to the chart. Negation would provide for uncontrolled descriptive imprecision, and is therefore disallowed. Further, the or–operator does not signify alternative specifications for a single chart section. Instead, the arguments of a disjunction in rhs appear on different sections, all of which are added to the chart in the proper place. Thus, for example the rhs <A:B> (<C:D>|<E:F>) would cause two strings to be added to the chart. One would contain <A:B> <C:D> and the other would contain <A:B> <E:F>.

(D15) lterm → [prefix] [¬] { lnode / [{#, @}] (lhs) / [¢] valvar } [lmod^n]

(D16) rterm → [prefix] { rnode / (rhs) / [¢] valvar } [rmod^n]

(D17) lmod → [prefix] { * / ? / varass / (lmod^n) }

(D18) rmod → [prefix] { ? / varass / (rmod^n) }

Both lterms and rterms correspond to what in conventional grammatical descriptions would be nodes or sequences of nodes, represented respectively by the two upper alternatives in (D15) and (D16). lnode and rnode are specifications of individual nodes, and will be described in detail below (Section 4.5). Admitting the parenthesized lhs and rhs allows Boolean expressions of lterms and rterms to be grouped together and given prefixes and modifiers as a unit. Notice that transformations may not appear within an rterm. lterm and rterm differ in that an lterm can be preceded by the logical negation operator and lmod includes an asterisk as well as a question mark. The negation is the negation from the Boolean expression, included here so that it may occur within the scope of a prefix; it is not allowed in rterm for the same reason that it does not appear in rhs.

The lmod gives a degree of imprecision to the lterm specification. The question mark, if present, signifies that the lterm containing it is optional, that is, that the rule can match either if the appropriate element of the chart matches the lterm, or else if the left side with the particular lterm deleted matches the chart. The asterisk indicates that the lhs containing the lterm may be interpreted as having a multiple number (1,2,3,...) of adjacent occurrences of the lterm. The symbols ? and * may appear more than once in lmod, but the effect of multiple occurrences is the same as that of single appearances.

Notice that all elements of lmod may be preceded by prefixes and that a prefix may also precede a number of lmods grouped together with parentheses. If an lmod contains both ? and *, it can occur zero or more times. Thus the lhses (40) and (41) are equivalent and will match any of the chart sections (42)—(44):

(40) <A:B> <C:D>*? <E:F>

(41) <A:B> <C:D>?**? <D:F>

(42) <A:B> <E:F>

(43) <A:B> <C:D> <E:F>

(44) <A:B> <C:D> <C:D> <C:D> <C:D> <E:F> etc.

If the lterm to which an lmod is appended is a parenthesized lhs, the lmod applies to the group as a whole. An asterisk would mean that the whole ordered group could appear a multiple number of times as a unit.

The asterisk is not allowed in rmod, again because the rterm must be an explicit description of what is to be created, with no arbitrary elements. The question mark may appear, however, in which case at least two new sections are added to the chart. Just as with rhs disjunctions, one section contains the questioned rterm and the other does not.

The symbols # and @, allowed in lterm, are not permitted in rterm because they also provide for ambiguities of specification which can be cleared up only by comparison with a real chart and not by constructing a previously undefined

chart section. Both of these symbols modify the way in which concatenated lterms in the parenthesizes lhs are interpreted. Normally, a sequence of lterms and a chart section will match only if there is an ordered one-to-one correspondence of lterms with matching chart modes. Thus the lhs <A:B> <C:D> <E:F> will match the chart section <A:B> <C:D> <E:F>, but will not match the section <A:B> <E:F> <C:D>. The # and @ alter this normal matching procedure. The # before a parenthesized lhs indicates that the lhs is to be interpreted "bidirectionally," that is, it will not only match a section of chart in the normal way, but also it will match a section of chart which would match the mirror-image of the lhs in the normal way. Hence #(<A:B> <C:D> <E:F>) will match both

(45) <A:B> <C:D> <E:F> (normally)

(46) <E:F> <C:D> <A:B> (mirror-image)

This facility therefore provides a straightforward way of expressing so-called "mirror-image rules" in natural language (Langacker, 1969).

The symbol @ means that the following lhs is to be interpreted as being "omnidirectional" or unordered. It will match a chart section which matches any permutation of the lterms of the lhs. Consequently, @ (<A:B> <C:D> <E:F>) will match

(47) <A:B> <C:D> <E:F> (normally)

(48) <A:B> <E:F> <C:D>

(49) <C:D> <A:B> <E:F>

(50) <C:D> <E:F> <A:B>

(51) <E:F> <A:B> <C:D>

(52) <E:F> <C:D> <A:B> (mirror-image)

The scope of the @ and # is limited to the top-level lhs enclosed within the parentheses. Thus if the top-level lhs contains an lterm which is also a parenthesized lhs, this second lhs will not be affected by @ or #. Its position among the other lterms in the top-level lhs will be modified, but its internal lterms will retain their original ordering. For example, #(<A:B> (<C:D> <E:F>)) will match chart sections (53) and (54) but will not match (55):

(53) <A:B> <C:D> <E:F>

(54) <C:D> <E:F> <A:B>

(55) <E:F> <C:D> <A:B>

This scope limitation has some peculiar consequences: #(<A:B> #(<C:D> <E:F>)) is equivalent to @(<A:B> <C:D> <E:F>), not # (<A:B> <C:D> <E:F>), for instance. Also, the lhs @ (<A:B> <C:D>*) will match any chart section that has <A:B> somewhere adjacent to or in the middle of an arbitrary number of nodes matching <C:D>. On the other hand, @(<A:B> (<C:D>*)) will match a chart section only if it contains a node matching <A:B> immediately before or immediately after a sequence of nodes matching <C:D>. Combinations of @ and # at different levels should be used very carefully.

The variable and prefix options, discussed more extensively in Sections 4.8 and 4.9, provide much additional flexibility. Briefly, prefix is a propositional function which, if it evaluates to FALSE, causes the lterm, rterm, or other element of which it is a constituent to be completely ignored by the parser. The parser skips to the immediately succeeding constituent and processes it instead. The varass and valvar alternatives enhance the power of the notation by permitting later references to previously identified and matched sections of the chart. When an element of the chart matches a rule constituent that has varasses appended to it then the chart element is tagged and associated with all the variables specified in the varasses. Later occurrences of valvars containing the same variables are references to the value of the variables, namely, that chart element. Notice that varass, just like the other constituents of lmod and rmod, may be preceded by a prefix, allowing conditional variable assignments.

Examples of lterm:

(56) <PS:NOUN_NUM:SG>?.N

(57) (<PS:VERB><PS:NOUN> | <PS:ADJ>).VP.PROP

(58) ¬#(<PS:NOUN>*.N ∉'VERB)?

Examples of rterm:

(59) <PS:NOUN>.N

(60) (<PS:VERB> <PS:NOUN> <PS:ADJ>)

(61) ('VERB? | ∉'NOUN <PS:SENT>).VP

4.5 Nodes

(D19) lnode → lspec [lsub]

(D20) rnode → rspec [rsub]

(D21) lspec → < [lbool] >

(D22) rspec → < [rbool] >

(D23) lbool → %BOOL[lelement, _, ¬]

(D24) rbool → %BOOL[relement, _, ¬]

(D25) lsub → { $ / ¢ } (lhs)

(D26) rsub → ¢ (rhs)

An lnode describes a node to be matched in the chart. This description consists of an lspec, an optional Boolean expression of lelements enclosed in angle-brackets, followed by an optional lsub. The lspec specifies conditions which the property list of the chart node must satisfy, while the lsub describes the subtree the node must dominate. A match will occur only if both the lspec and lsub descriptions are satisfied. The list of attribute-value pairs of a node in the chart will satisfy an lspec if it contains a subset which will make the lspec TRUE. Each lelement is assigned the value TRUE if and only if either an appropriate pair occurs in the list, or, if the lelement begins with ¬, an appropriate pair does not occur. The lspec is then assigned a truth-value by evaluating the Boolean expression in the normal manner. Notice that the and-operator is the underscore and represents logical conjunction, not concatenation.

Thus, the truth-value finally assigned will be independent of the order in which the lelements are written, and it is possible for a single pair to satisfy more than one lelement. The order can affect variable assignments and prefix evaluations, however. Notice also that ¬X, where X is a combination of lelements, will be TRUE only if X cannot be satisfied and not if something which is not X is encountered. Therefore, varasses within X will not be executed. This procedure will always assign TRUE to the empty lspec so that a pair of angle brackets enclosing nothing will match every attribute-value list, while its negation will not match any property list. The lterm <>*? thus behaves like the free variable in transformational theory.

An rnode is structurally similar to an lnode. The rspec describes the attribute-value list to be inserted in the node being added to the chart, and the rsubtree defines the structures which the new node is to dominate. An rspec can also be empty, so that the chart can contain nodes with no attribute-value pairs. Because of the procedure by which lspecs are satisfied, such nodes will only satisfy lspecs which are empty or which are of the form ¬X, where X is a combination of lelements. Instances of sub-expressions within an rspec can be separated into two disjoint categories, those expressions which comprise the scope of a negation but are not within the scope of any other negation, and those expressions which are not within the scope of any negation

at all. Thus in the rspec <A:B_¬(C:D|¬E:F)_G:H>, (C:D|¬E:F) is in the first category while A:B and G:H are in the second. Note that C:D and ¬E:F are in neither category, even though they are well-formed Boolean expressions. relements in expressions in the second category specify the attribute-value pairs to be added to the chart. Within these expressions the relements are processed in the left-to-right order in which they appear; the arguments of the and-operators are added to a single node, and the arguments of the or-operators are added to separate, alternative nodes. Like the or-operator in rhs, the or-operator here signifies the addition of more than one new section to the chart. If there is a conflict[1] between conjoined relements, then the one processed last (the right-most) overrides all the others. Examples:

(62) <A:B_C:D_E:F>

(63) <A:B_'X_E:F>

(64) <A:B_(C:D|E:F)>

The rspec (62) instructs the parser first to add A:B to the appropriate node, then C:D, and finally E:F. (63) causes the parser to add A:B, then all the pairs contained in the valvar 'X, and then E:F. If the pair E:G appeared in 'X, the later occurrence of E:F would cause the final result to contain E:F, not E:G. On the other hand, if 'X contained

[1]Since an attribute can have only one value in a single chart node, a conflict exists when two different values are to be given to the same attribute.

A:H, the final result would include A:H instead of A:B. (64) will generate two new strings in the chart, one of which will have a node containing A:B_C:D while the other has a node with A:B_E:F.

Sub—expressions in the first category, those which comprise the scope of a negation operator, are prescriptions for removing pairs which were added to the node in the processing of second—category expressions further to the left in the rspec. Of course this facility is necessary only to remove pairs which were implicitly added to the node through variable references: the user would never explicitly add unwanted pairs. The parser interprets a second—category expression in the following way: the expression following the prefix negation is processed as an lspec and is compared to the portion of the new node already constructed through the action of preceding parts of the rspec. Attribute—value pairs in the node which are associated with relements in the expression and thereby make it TRUE are deleted. varasses in the expression are executed as they would be in an lspec. If the truth—value of the expression is FALSE, it is completely ignored by the parser, and processing skips to the immediately succeeding expression. In the rspec (65) if 'X contains E:F, the pair will be eliminated by ¬E:F.

(65) <A:B_'X_¬E:F>

(66) <'Y_P:Q_¬((R:S|T:U)_¬V:W)>

In (66), if 'Y includes R:S or T:U or both, they will only be deleted if 'Y does not also contain V:W.

If an lspec matches the attribute-value list of a node in the chart, the parser tests to see if the lsub of the lnode is also satisfied. lsub is a parenthesized lhs preceded by one of the symbols $ or ¢. The lhs matches nodes in the chart in the manner described above; the symbols indicate to the parser which nodes in the chart are to be examined. They both require that the nodes be in the subtree of the node which matches the lspec and that they must comprise a proper analysis[2] of the subtree. The dollar-sign signifies that an arbitrary proper analysis will do, while the cent-sign indicates in addition that the sections of chart to be considered must be immediate daughters of the node.

---

[2]The notion "proper analysis of a subtree of a node in a tree" may be defined as follows: Let H be a tree and let $A, X_1, \ldots, X_n$ be nodes in H. For every node Y in H define the set

$$T(Y) = \{x \mid (x \text{ is a terminal node of H}) \ \& \ (Y \text{ dominates } x)\}$$

Then the sequence of nodes $X_1, \ldots, X_n$ is a proper analysis of the subtree dominated by A if and only if:

(1) $\forall i$(A dominates X_i)

(2) $\forall i \forall j (T(X_i) \cap T(X_j) = \emptyset)$

(3) $\bigcup_{i=1}^{n} T(X_i) = T(A)$

(4) $\forall i \forall j \forall a \forall b (i<j \ \& \ a \in T(X_i) \ \& \ b \in T(X_j) \supset$ a precedes b in H)

rsub delineates the subtree to be constructed under a node. Here only the cent-sign is allowed: the dollar-sign does not precisely determine the levels at which the nodes in the subtree are to appear. The immediate daughters specified in rsub also make up a proper analysis of the subtree in the chart. Subtrees of any depth may be created by building downwards node by node (Cf. (71)). Examples of lnode:

(67) <A:B_C:D> $(<Z:W> ¢(<>* <Q:R>))

(68) <A:B_C:D_(¬E:F|'VAR)_'VAR3|(Z:W_P:Q)>

(69) <PS:NOUN_(NUM:PL|NUM:SG)_GEN:FEM>

(70) <PS:ADJ|PS:NOUN_(GEN:MASC|GEN:NEUT)>

Examples (67) and (68) demonstrate the power of the notation, while (69) and (70) represent possible uses in actual grammatical rules. Examples of rnode:

(71) <A:B>¢(<X:Y>¢(<P:Q>¢('V <R:S_T:V> <U:W>)))

(72) <'NP1_¬GEN:FEM_NUM:SG>¢(¢'NP1)

(72) is a noteworthy application of the notation: it would cause a node to be created with all the attribute-value combinations of NP1, except that it would have singular number but not feminine gender. The cent-sign before the quote in the rsub means that the subtree of the new node is to be the same as the subtree under NP1 (Cf. Section 4.8). This is one way in which attribute-value pairs can be altered without

changing the subtree of a node.

(D27) lelement — [prefix] [¬] $\left\{ \left\{ \begin{matrix} \text{valvar} \\ (\ \text{lpair}\ ) \end{matrix} \right\} [\text{varmod}^n] \right\}^{\text{lpair}}$

(D28) relement — [prefix] [¬] $\left\{ \left\{ \begin{matrix} \text{valvar} \\ (\ \text{rpair}\ ) \end{matrix} \right\} [\text{varmod}^n] \right\}^{\text{rpair}}$

Structurally, lelement and relement are similar. Their central constituents are lpair and rpair, respectively. lpair characterizes a group of attribute–value pairs, and if at least one member of this group is identified in the appropriate chart node, the lelement will be evaluated TRUE. rpair precisely defines a single attribute–value pair to be inserted in a node. The varmod options, denoting sequences of varasses, mean that pairs as well as nodes and sequences of nodes can be assigned to variables, either when they are matched on the left or constructed on the right. Notice that lpair and rpair must be enclosed in parentheses when they are followed by varmods. If a pair is associated with a variable, the valvar alternative can be used to represent that pair later on in the same rule.

The prefix option again signifies that the appearance of the elements in the rule is conditional on prior variable assignments. If the prefix is FALSE, the negation

operator is disregarded along with the rest of the lelement or relement.

4.6 Attribute-Value Pairs

(D29) lpair → literal $\left\{\begin{matrix}:\\!\end{matrix}\right\}$ $\left\{\begin{matrix}\text{lspec } [\text{varass}^n]\\ \%\text{SET}[\text{lpair},\text{prefix},-,\text{varmod}]\end{matrix}\right\}$

(D30) rpair → literal : $\left\{\begin{matrix}\text{rspec } [\text{varass}^n]\\ \%\text{SET}[\text{rpair},\text{prefix},\emptyset,\text{varmod}]\end{matrix}\right\}$

Basically a pair consists of an attribute literal, a separator, and then a value which can be either a specification or a value-set. The specification alternative is provided so that attribute-value pairs can be logically grouped together. For example, if SF is a literal representing "selectional features," then the following lhs would match a chart section only if the selectional features of the noun satisfied the verb's subject selectional restrictions:

(73) <PS:NOUN_SF:<>.NSF> <PS:VERB_SUBJECT:<SF:<'NSF>>>

There are two procedures by which an lpair is given a truth-value, depending on the separator. If the separator is a colon, then the parser searches the attribute-value list of the chart node for the attribute specified by the literal, and if the attribute does not appear in the list, the lpair is immediately FALSE. If the required attribute does appear, then its value is examined. If the lpair value and the value of the attribute in the chart are not both either specifications or sets, then the lpair is FALSE. Otherwise, if they are both specifications, the lspec is

compared with the list of attribute-value pairs comprising the value of the matched attribute in the chart, just as lspec is processed at the higher level of lnode. If the lspec is TRUE, then the lpair is TRUE and the specification in the chart is associated with the variables designated in the varmods. If the lspec is FALSE, the lpair is FALSE and no varmods are executed. If the lpair value and the value of the matching chart attribute are both sets, then the two sets are intersected. If the intersection set is non-empty, the lpair is TRUE and the intersection set is assigned to the appropriate variables. If the intersection set is empty, the lpair is FALSE and, as above, no varmods are processed.

The procedure for assigning truth-values to lpairs is slightly different when the separator is an exclamation mark. In this case, an lpair will be TRUE in two circumstances: first, if it is TRUE by the procedure outlined above for the colon separator, and second, if it is FALSE by this procedure only because the attribute specified by the literal does not occur at all in the chart attribute-value list. If the literal ANY has the internal representation 0 and so denotes the universal value set, the relationship between the colon

and the exclamation mark is expressed in the following theorems: if L is a literal, S is an arbitrary lpairset, and <P> is an arbitrary lspec, then:

(74) L!S ≡ L:S|¬L:ANY

(75) L!<P> ≡ L:<P>|¬L:< >

In both (74) and (75) if the first alternative is not satisfied but the second is, the lpair is TRUE but all varmods are distegarded. With the exclamation mark, then, an lpair is FALSE only if the lpair and chart have the same attribute with different values. This is the condition for a "violation" in the sense of Lakoff (1965), and this notation makes it possible to capture the distinction between major and minor rules, with the resulting simplification of the lexicon. Suppose, for example, that Rule 79 is a major rule governed by verbs, and that the verb V is an exception to the rule. In the left side of Rule 79 the lnode <PS:VERB_R79!PLUS> would appear. This rule would apply to all verbs which are not marked for the feature R79, but an exception such as V would have a chart marked <PS:VERB_R79:MINUS>, and thus the rule could not apply to V.

Now consider Rule 80 to be a minor rule, that is, a rule which regularly does not apply but, instead, applies only to exceptions. The left side of this rule would have the lnode <PS:VERB_R80:PLUS>, and if V were an exception requiring the rule to apply, it also would be marked <PS:VERB_R80:PLUS>. All other verbs would be unmarked and Rule 80 would not apply to them. Besides allowing for simplification of the lexicon and the implementation of major and minor rules, rule features enable the user to establish explicit ordering relations among the rules apart from the implicit restrictions available through id numbers.

The internal structure of %SET is defined in Section 3.2. Recall that the commas separating singletons denote ordinary set union: A,B,C is the set consisting of the union of the literals A,B, and C. Singletons in both lpair and rpair sets can begin with a prefix which, if it evaluates to FALSE, signifies that the following constituent is to be ignored by the parser. If the processing of prefixes results in the elimination of a whole value-set, then the complete pair is disregarded. (If all pairs of a value specification are eliminated, then it is processed just as <> is ordinarily processed at the node level).

The minus-sign can occur on the left but not on the right. The minus-sign represents the complementation operation: the set -A designates the complement of A relative to the literal universe. Thus the intersection of -A with a chart set is simply the complement of A relative to that chart set, and in particular, the intersection

of -ANY with any chart set is empty. A singleton is fundamentally a literal, but it can also be realized as a parenthesized set. This alternative is included so that the scope of prefixes, minus-signs, and varmods can extend over more than one singleton. The parentheses have no effect if none of the options are chosen. Hence, L:(A,B,C) is exactly equivalent to L:A,B,C. When the minus-sign appears before an lpairset, the expression denotes the complement of the entire set relative to the universe. Notice that the minus-sign and the negation operator impose different conditions for a successful match: Q:-A requires a chart node with an attribute Q, while ¬Q:A does not. The minus-sign can often lead to redundancy; the lpairset (A,-(B,C)) is equivalent to -(B,C) for example. In general, complements within unions make sense only in conjunction with prefixes or when valvars are the lpairsingletons.

The valvar option provides the capability of recalling a set that was previously matched in the chart. If a valvar is preceded by the xymbol #, it designates the set consisting of the union of the opposites of all the literals in the set represented by the valvar. (Remember that the opposite of a literal is defined in a declaration at the beginning of the grammar.) This facility provides a simple notation for expressing the commonly used linguistic device called "alpha switching:" if X is a variable and 'X is SING, #'X might be declared to be PLUR. The lhs <NUM:ANY.X> <NUM:#'X> would only match two chart nodes with opposite number specifications.

lpair examples:

(76) A:B.Y

(77) 25:#'Y

(78) L:¬(D,E)

(79) A:<B!<C:D_E:F_G!H>.1>.2

(80) A!<(Q:R|F:G)_¬(X:Y_Z:W)>

An rpair looks very much like an lpair. It is distinguished only by the absence of the exclamation mark as a separator and of the minus—sign in the set. These restrictions are consistent with the general constraint that the user must have definite positive knowledge about what is to be put into the chart, and hence must be able to enumerate explicitly the elements of the rpairsets. A small, controlled degree of uncertainty is provided by prefixes here as elsewhere in rhs. Notice that in general, the set value of an attribute added to the chart is a union of literals and not just a single literal. (76) and (77) are rpairs as well as are (81) and (82):

(81) X:'Q

(82) 1:<X:Y_Z:W>.N

4.7 Part—of—Speech Abbreviation

In current grammatical descriptions, syntactic rules are stated primarily in terms of the values of a single attribute, the part—of—speech or syntactic—category attribute. Even in grammars where complex symbols are allowed (Chomsky, 1965),

the syntactic categories have a privileged position in the rules. In MGRL, however, part-of-speech has the same status, in the syntax as well as in the semantics of the rules, as any other attribute: the attribute literal must precede all its values in node specifications, and the user, through a declaration, can assign any internal code to it, although the internal code 1 is recommended. The frequent repetition of the attribute in the specification of different nodes lengthens and complicates the rules, often unnecessarily since the user's intentions are clear. Thus it is desirable to extend MGRL to permit an abbreviation of one attribute, to allow the attribute to be given implicitly. This means that, without sacrificing any of the power of the language, rules written in MGRL can be greatly simplified and made to resemble more closely rules expressed in more conventional forms. Although the user can give any significance to the ommitted literal, the parser will always assume that its internal code is 1.

The abbreviation facility is provided in MGRL by replacing (D19)-(D20) with (D19')-(D20') and adding the meta-rules (D21a)-(D22a):

(D19') lnode → lspecabr [lsub]

(D20') rnode → rspecabr [rsub]

(D21a) lspecabr → { [+] lspec
                  { + %SET[lpair,prefix,-,varmod] [lspec] }

(D22a) rspecabr → { [+] rspec
                  { + %SET[rpair,prefix,∅,varmod] [rspec] }

A node can thus be realized as a set followed by an optional specification and an optional subtree description. The set must be preceded by a plus-sign, and it is processed as the value of the implicit attribute. Thus, if PS has bit-code 1, the lnode (83) is equivalent to (84):

(83) +NP<NUM:SING>

(84) <PS:NP_NUM:SING>

Notice that the parser processes the abbreviated attribute-value pair as if its separator were a colon and as if it were the first pair of the specification. Hence, the convention that varmods are executed in the order in which they appear still holds. If the node begins with a set, it need not also contain a specification: if the specification is omitted, the node is interpreted as if the set were succeeded by the empty specification. Hence, (85), (86), and (87) give the same result:

(85) +VP,ADJ

(86) +VP,ADJ<>

(87) <PS:VP,ADJ>

If the set is followed by a specification which already has an element with attribute PS, the parser does not combine the set with the value of the explicit attribute in any way; it recognizes two separate pairs that just happen to have the same attribute. Such an lnode, for example, will match a chart node only if the chart value-set for the attribute with internal code 1 matches each of the lelements independently.

(88) and (89) are equivalent:

(88) +VP<PERSON:THIRD_PS:ADJ>

(89) <PS:VP_PERSON:THIRD_PS:ADJ>

The abbreviation facility thus makes MGRL rules easier to write and easier to read, but, in a sense, it goes against the spirit of the rule language as a whole and therefore requires special rules of interpretation. Apart from the abbreviation, MGRL is defined so that any syntactically well—formed statement will be interpreted in one and only one way by the parser. However, since a node can now be realized as a single set outside a specification, it is possible for one rule to be generated in two different ways. Consider an lhs containing two adjacent lnodes, the first one being realized as a set without a following lspec, lsub, or lmod, and the second one beginning with an lspec, as in (90):

(90) +VP <NUM:SING>

Under this interpretation, (90) should be processed identically to (91):

(91) <PS:VP> <NUM:SING>

However, since spaces are disregarded by the parser, a single lnode consisting of a set followed by an lspec would also appear as (90), but should be processed as (92):

(92) <PS:VP_NUM:SING>

On the right also the same rule configuration can be generated in two ways. The ambiguity is resolved in both cases by the following rule of interpretation: the parser will always interpret a set and

a following specification or subtree separated from it only by spaces as a single node. Thus (90) will always be processed as (92), not (91). The optional plus—sign in the first alternatives of (D19') and (D20') must be inserted if the user intends the interpretation illustrated in (91). The user must rewrite (90) as (93):

(93) +VP +<NUM:SING>

The plus—sign before a specification is not really optional in the sense that its presence or absence makes no semantic difference. It is optional at the beginning of a node not containing a set only if the preceding node ends with a property or subtree specification, or an lmod or rmod. If the preceding node is a simple set, the plus—sign is mandatory. This requirement essentially undermines the context—free structure of MGRL, and should properly be stated as a context—sensitive meta—rule. This complication of the rule language is justified by the usefulness of the abbreviation, but because of it, the abbreviation should be used cautiously.

The abbreviation engenders one other ambiguity: when the parser establishes by the interpretation rule above that a node consists of a solitary set, then varasses following the set could be processed either as varmods applying within the set or as lmods or rmods applying to the node as a whole. (94) could be interpreted as (95) or (96):

(94) +A,B,C.V +X

(95) <PS:A,B,C.V> <PS:X>

(96) <PS:A,B,C>.V <PS:X>

This structural ambiguity is also resolved by an interpretation rule. The parser will always try to interpret the varass at the node level first; only if the rule is not well-formed under that interpretation will the set-level interpretation be considered. If the user intends the varass to apply within the set, he must enclose the varass within set-level parentheses. (94) will always be processed as (96); for the interpretation (95), the user must write (97) or (98):

(97) +A,B,(C.V) +X

(98) +(A,B,C.V) +X

Thus again the interpretation rule required by the abbreviation is equivalent to the addition of a context-sensitive restriction to MGRL.

It should be noted that the pair-level options permitted within a specification are not allowed. The negation-operator cannot apply to an abbreviated pair, pair-level varmods cannot occur, and prefixes are not permitted. If any of these options is necessary, the pair should be specified in its expanded form. However, if the whole set of the abbreviated pair is to be disregarded, then the effect is the same as would be obtained by a pair-level prefix, according to the convention mentioned in Section 4.6.

4.8 Variables

A variable is a device for identifying and referring to previously processed sections of the chart. Like variables in most high-level programming languages, a variable might be thought of as designating a storage location. The execution of a varass causes a part of the chart to be inserted into the storage area named by the variable contained in the varass, thereby becoming its "value." In a left side the section of chart assigned to a variable is the section that matches the constituents immediately preceding the varass; on the right, the value of a variable is the chart section created according to the specifications of the immediately preceding constituent. The contents of a variable storage location are recalled through valvars containing the variable.

(D31) varmod → [prefix] { . variable / (varmod^{n}) }

(D32) varass → . variable

(D33) valvar → ' variable

Values of variables can be nodes and sequences of nodes, pairs, and sets. Notice that varasses only occur as constituents of lmod, rmod, or varmod, so that they can be grouped together and preceded by a prefix. Thus the assignment of variables can be conditional on previous variable values. Since a sequence of varasses is always allowed, a given chart section can be the value of more

than one variable. On the other hand, more than one chart section can be the value of a given variable. Consider the following rule body:

(99) (<A:B>.V1.V2 <C:D> <E:F>.V1).V3 = 'V3 'V2 'V1

Suppose that in the chart the nodes x,y, and z match the lnodes <A:B>, <C:D>, and <E:F>, respectively. Then if (99) is compared to the chart section x y z, the node z will be in the value of both V1 and V2, x and z will both be associated with V1, and the whole section x y z will be the value of V3. A multiple assignment to a single variable at the node level results in the variable's value being the concatenation of all the assigned nodes in the left—to—right order in which they are matched. Thus (100) and (101) are equivalent:

(100) (<A:B> <C:D>).V

(101) <A:B>.V <C:D>.V

In light of this, the rule body (99) will add the section x y z x x z to the chart corresponding to the old section x y z.

When two or more pairs are assigned to the same variable, the value of the variable is also derived by concatenating the pairs in the order in which they are encountered. This order is disregarded when a valvar occurs on the left, but, as usual, it is significant when a valvar in rspec is processed. (102) and (103) will therefore have the same effect, but the effect of (104) is potentially

different.

(102) <(A:B).V1> <(A:B,D).V1> <'V1> = <'V1>

(103) <(A:B).V1> <(A:B,D).V2> <'V2_'V1> = <'V1_'V2>

(104) <(A:B).V1> <(A:B,D).V2> <'V2_'V1> = <'V2_'V1>

The general convention is that multiple assignments to a single variable at one of the three levels make the value of the variable the conjunction, interpreted according to the level, of the various assigned chart sections. For sets, the appropriate and—operator is intersection, and the outcome of (105) will be the same as the outcome of (106):

(105) <A:ANY.X> <B:ANY.X> = <C:'X>

(106) <A:ANY.Y> <B:ANY.Z> = <C:—(—'Y,—'Z)>

The notation of (105) is clearly simpler than that of (106) and correspondingly, (105) is processed more efficiently. Notice that the set value of a variable can become, through multiple assignments, the null set. If the set value of V is empty, L:'V is equivalent to ¬L:ANY on both the left and the right.

It is also possible to assign values to a single variable at more than one level. In (107), the variable X is associated with chart sections at all three levels:

(107) (<A:ANY.X> <(B:Q,R).X>).X = <C:'X_'X> 'X

The parser processes this as if there were actually three distinct variables: X_s at the set level, X_p at the pair level, and X_n at the node level. Multiple assignments within a given level are derived in the normal manner for

that level. Since the level at which either a valvar or a varass occurs can always be determined from context, the conceptual distinction between the different levels is consistently maintained by the parser. In actual operation, (107) would be equivalent to (108):

(108) $(\langle A{:}ANY.X_s\rangle \langle (\ B{:}Q,R).X_p\rangle).X_n = \langle C{:}'X_s\_'X_p\_'X_n\rangle'X_n$

The $'X_n$ in the rspec illustrates an important principle, namely, that the pair value of a variable is more than just the concatenation of the pairs explicitly assigned to it. It includes in addition the complete property lists of all the chart nodes assigned to the variable. The current pair value is formed by conjoining the pair assignments with these property lists in the order in which the relevant varasses are encountered. Accordingly, in (107) the final pair value of X will have the pair matching B:Q,R preceding the pairs implicitly assigned by the node-level varass, and hence, $'X_p$ precedes $'X_n$ in the rspec in (108). Of course, within a property list the order of pairs is fixed by the internal representations of the attributes, but because an attribute can have at most one value per chart node, the operation of the parser is logically independent of the arrangement of pairs within nodes. It is therefore unnecessary to assign to a variable pairs contained in the last node assigned to the variable: such assignments will only cause the parser to work harder with no substantive effect. Rewriting (108) and (109) will not alter the

result:

(109) $(\langle A{:}ANY.X_s\rangle\ \langle B{:}Q,R\rangle).X_n = \langle C{:}'X_s\_'X_n\rangle\ 'X_n$

Notice that with this definition of the pair value of a variable, a variable associated with a single node can be used either at the node level to refer to the node and the subtree dominated by it, or at the pair level to refer just to the attribute-value list of the node.

Reference to the values of a variable is made through a valvar containing the variable. The level at which the valvar occurs determines which of the three separate variable values is to be recalled. The effect of a valvar is usually different from the effect that would be obtained by repeating the rule constituents which had had the relevant varasses attached. On the left a valvar imposes more stringent matching conditions than those imposed by a repetition of constituents, while on the right the recipe for the new chart section is specified in greater detail. For example a node-level valvar not only requires for a match nodes with property lists identical to those of the nodes in the variable value, but also requires node-by-node identity in the subtrees. On the right a valvar results in the insertion into the chart of exact copies of the nodes in the variable value, including the subtrees they dominate. Similarly, a set valvar will match a chart set only if the intersection of the chart set with the current value of the variable is non-empty; hence a mismatch might

occur even if the chart set matches all the lpairset descriptions associated with the variable. The set that results from a valvar is the intersection of various sets, some from the chart and some from the rule, and in general cannot be described by explicit rule constituents.

An additional powerful option is available at the node level. valvars in lterm and rterm may be preceded by the symbol ¢. The cent-sign in a regular node description denotes the string of immediate daughters of the top node, and before a valvar it also designates immediate daughters: ¢ valvar represents the string of subtrees obtained by concatenating all the immediate daughters of all the nodes in the value of the variable. Accordingly, if the node-value of the variable X is a sequence of two nodes, the first of which dominates <A:B> <C:D> while the second dominates <E:F>, then ¢'X refers to the string of subtrees dominated by the nodes <A:B> <C:D> <E:F>. The ¢ valvar operation is a particular instance of the more general process specified by the transformation REMOVE (cf. Section 4.10).

What happens when a valvar appears in a rule before the variable has been assigned a value? If a variable does not yet have a value at the level at which the valvar occurs, then the variable is said to be "undefined" at that level, and the valvar is simply ignored by the parser. This procedure can be formally represented by inserting before each valvar at a given level a prefix which will be

TRUE only if the variable has been defined at that level. Thus if X is a variable and if X_n, X_p, and X_s denote its node, pair, and set value components, then valvars involving X are processed as if they occurred in the expressions $/(X_n)'X_n$, $/(X_p)'X_p$, and $/(X_s)'X_s$, respectively. (For any variable Y, /(Y) is a prefix which is FALSE only if Y has not been assigned a value. Cf. Section 4.9.) Recall that one additional convention has been adopted: if the whole set value of an attribute is to be ignored because prefixes are FALSE, then the parser disregards the whole pair.

4.9 Prefixes

The parser will ignore all rule constituents within the "scope of a prefix" if the prefix is FALSE. The scope of a prefix includes all constituents which are defined in the metalanguage as right-sisters of the prefix. Often the scope will extend only to the constituent immediately succeeding the prefix, but this is not always the case. In an lterm, for example, the lmod is within the scope of the initial prefix. The internal structure of prefix is as follows:

(D34) prefix → / (%BOOL[condition,&,¬])

(D35) condition → { variable / valvar = %SET[prefix,∅,∅,∅] }

A prefix is set off by a slash and its length is indicated by the parentheses. It consists of a Boolean combination of conditions. The parser assigns a truth-value to a condition in one of two ways: a simple variable condition

will be TRUE only if the variable has been defined at least at one level. An equality condition will be TRUE if and only if two situations obtain: (1) the variable specified in the valvar has a set value, and (2) the intersection of the set value of the variable with the set described by the prefixset is non—empty. A prefixset is similar to an lpairset or rpairset, except that it may not contain pre—fixes, minus—signs, or varasses. Examples of prefix:

(110) /(NP&VP)

(111) /(NP&'GEN=MASC,FEM)

(112) /(NOUN|¬VERB)

(110) will be TRUE if both VP and NP have been assigned values. (111) will be TRUE if NP has a value and if the set value of GEN contains MASC or FEM or both. Finally, (112) will be TRUE if either NOUN has a value or VERB does not.

prefix is one of the most powerful devices in the rule language since it provides the facility for expressing cross—serial dependencies between the parser's actions at various points in a rule. For example, the lhs of a rule might contain an alternation of lnodes, each one containing a varass with a different variable. The rhs can then be divided into subsections, one subsection for each alternative, by prefixes in such a way that only the sections corresponding to the alternatives that were successfully matched on the left will be executed. By logically extending this procedure a

whole grammar could be condensed into a single rule. However, the normal use of prefixes is to conflate two or more rules that differ only in minor details.

4.10 Transformations

(D36) transformation → tname (variable $\left\{\begin{matrix}[, \text{variable}]^n \\ [, \text{rterm}]^n\end{matrix}\right\}$)

(D37) tname → character^n

The tname designates a certain elementary operation to be performed by the parser on the section of the chart created according to the specifications of the right side constituents to the left of the transformation. The variables appearing as arguments of the transformation identify and "locate" particular nodes already in the new chart section while the rterms describe insertions to be made at appropriate places in the chart. An argument variable locates all chart nodes for which either of the following situations obtains: (1) the nodes are actually contained in the value of the variable because the rterms by which they were constructed have varasses containing the variable appended to them. These rterms may have appeared either in the rhs or as arguments to previously executed transformations, that is, transformations appearing farther to the left in the rule. (2) the new chart nodes are copies of nodes in the value of the variable. These copies will appear in the chart if a rterm to the left of the transformation

has a valvar containing either the argument variable itself, or another variable which includes in its value a node which matched the root of an lsub dominating a node assigned to the argument variable. In this last case, the valvar causes a copy of the whole subtree to be inserted in the chart, so of course a copy of the node assigned to the argument variable will be constructed. In the rule bodys (113) – (117), the variable V locates the new chart node corresponding to <A:B>.

(113) <C:D> = <A:B>.V "T(V);

(114) <C:D> = <X:Y>.U "T1(U,<A:B>.V) T(V);

(115) <A:B>.V = 'V "T(V);

(116) <A:B>.V = <X:Y>.U "T1(U,'V) T(V);

(117) <X:Y>¢ (<Q:R>$(<A:B>.V)).U = 'U "T(V);

The definition of "locate" implies that the order in which transformations are written can be important: if T(V) causes nodes located by V to be deleted, then T(V) T1(U,'V) is equivalent to T(V) but not necessarily to T1(U,'V) T(V). Notice that for a variable to locate a node, it must necessarily be node-defined, but the nodes located by the variable are not necessarily included in the variable's value.

The particular operations denoted by transformations are not defined in the metalanguage: their definition and any special restrictions on their arguments are left to the implementation of the parser. However, in any implementation certain operations will most likely be desirable. A number of these are adjunction transformations:

(a) RSA (variable , rterm)

RSA adds its second argument as a right sister to each of the nodes located by its first argument.

(b) RDA (variable , rterm)

RDA adds its second argument as a right daughter to all the nodes located by the variable.

(c) RCA (variable , rterm)

RCA right-Chomsky-adjoins its second argument to each of the nodes located by its first argument.

The implementation will also include, for each of the above, an equivalent left-handed transformation: LSA, LDA, and LCA, respectively. If the first argument of an adjunction transformation does not locate any nodes, or if the second argument contains a valvar that is not node-defined, then the transformation is ignored.

The following transformations have an arbitrary number of arguments, each of which is a variable:

(d) DELETE (variable [, variable]^{n})

DELETE causes all occurrences of all the nodes located by its arguments to be deleted. The subtrees dominated by these nodes are also eliminated. An argument not locating any nodes is disregarded.

(e) REMOVE (variable [, variable]^{n})

REMOVE, like DELETE, causes the nodes located by its arguments to be deleted. However, their subtrees are preserved: the daughters of the deleted nodes become daughters of the parents of those nodes.

Finally, the following transformations impose more idiosyncratic restrictions on their arguments:

(f) PERMUTE (variable , variable)

PERMUTE causes the nodes located by its arguments to be interchanged in the chart. Both arguments must locate single nodes: if an argument locates more than one node, the transformation operates just on the right-most located node in the chart.

(g) INSERT (variable , rterm)

A copy of the node specified by the second argument is introduced in the chart as the daughter of the parent of the node located by the first argument. The located node becomes the daughter of the inserted node. The rterm must specify a single node without a subtree. If it is realized as a valvar, a copy of the last-assigned node-value of the variable is inserted, but its subtree is ignored. INSERT is in some sense the inverse of REMOVE.

(h) RENAME (variable , rspecabr)

RENAME gives a new set of attribute-value pairs to the nodes located by the first argument, namely, those given as the second argument. The original pairs of the nodes are simply deleted.

transformations add to the power of MGRL in two principal ways: first, they enable operations to be performed

deep within subtrees without requiring that the subtrees be specified in detail. Second, they build into the MGRL syntax a means of invoking any sort of syntactic or semantic procedure. For example, a transformation could be implemented which would resolve semantic ambiguities by referring to a store of encyclopedic knowledge. Procedures of this type are being planned for MIND.

5. ILLUSTRATIVE RULES FOR ENGLISH

This section provides examples of plausible English grammar rules as they might be translated into MGRL. The examples are presented in three groups: the first rules are derived from the context-free base component of a generative grammar (Lakoff, 1967), rules in the second set correspond to the conventional transformations of that grammar, while rules in the third group have been extracted from an experimental recognition grammar being developed for the MIND system. It should be remembered that these examples are given only to demonstrate the various syntactic and semantic features of MGRL in a realistic context and that these particular rules most likely would not appear in an optimum English recognition grammar for the MIND parser.

5.1. Context-Free Phrase-Structure Rules

These MGRL rules parallel those in the base component of the Lakoff grammar. The MGRL rules are translations of the inverses of the generative rules, however, and a complete grammar would create a deep-structure phrase-marker for an English sentence, given the underlying terminal string with the appropriate syntactic specifications. In each example, the first line is the original generative rule, included for comparison, and the subsequent lines are alternative MGRL translations. The ordering relations expressed in the id fields are only illustrative and would

almost certainly be different in an actual grammar.

(118) (a) $S \rightarrow \begin{Bmatrix} and \\ or \end{Bmatrix} S^n \quad (n \geq 2)$

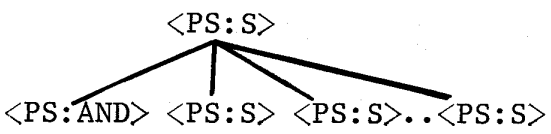
(b) 1, ((<PS:AND>|<PS:OR>)<PS:S> <PS:S>*).VS = <PS:S>¢('VS);

(c) 1, (<PS:AND,OR> <PS:S> <PS:S>*).VS = <PS:S)¢('VS);

(d) 1, (+AND,OR +S +S*).VS = +S¢('VS);

(118) describes the underlying structure of conjoined sentences. (118b) is the most direct translation of the inverse of (118a). It would result in the construction of a chart tree resembling (119a) corresponding to the string of trees (119b):

(119) (a) (b)



<PS:AND> <PS:S> <PS:S>..<PS:S>

If it were known that the literals AND and OR could not both occur in a single value set, then the disjunction in (118b) could be replaced by the union in (118c). The notation is simpler and (118c) would be processed more efficiently. (118d) demonstrates the part-of-speech abbreviation but is otherwise exactly equivalent to (118c), assuming that the declaration 1(PS) appears in the grammar.

(120) (a) S → (PreS) NP AUX VP $(ADV)^n$

(b) 2, (+PRES? +NP +AUX +VP +ADV?*).VS = +S¢('VS);

(c) 2, (+PRES? +NP +AUX +VP +ADV?*).VS = +S.T " LDA(T,'VS);

(120) deals with the constituents of single sentences. Notice that the optional sequence of adverbs is represented in MGRL by the question—mark and asterisk, and that the tree—building operation is denoted in (120b) by a varass and a valvar, just as in (118). This is a standard way of representing the inverse of a context—free production. (120c) achieves the same effect with the explicit daughter—adjoining transformation.

The verb—phrase expansion rule of the generative grammar (121a) is very ungainly in the conventional notation, and its direct translation into MGRL is no less awkward. However, the prefix feature of MGRL can be used to express the complicated cross—serial dependencies of the generative rule in a simple, straightforward way:

(121) (a)

$$\text{VP} \rightarrow \left\{ \begin{array}{l} \text{V} \quad (\{\text{NP}\ \text{PP}\}) \quad (\{\text{NP}\ \text{PP}\ \text{S}\}) \\ \text{BE} \quad \left(\begin{array}{l} \text{ADJ} \quad (\{\text{NP}\ \text{PP}\ \text{S}\}) \\ \{\text{NP}\ \text{PP}\} \end{array} \right) \end{array} \right\}$$

(b) 3//2, ((+V (+NP|+PP)? (+NP|+PP|+S)?) | (+BE (+ADJ (+NP|+PP|+S)?)|(+NP|+PP))).VVP = +VP¢('VVP);

(c) 3//2, (+V.VV,BE +/(¬VV)ADJ.A,NP,PP/(VV)? /(VV|A)+NP,PP,S?) .VVP = +VP¢('VVP);

In (121c) the disjunctions have been realized as set unions, and the three main paths through (121a) have been marked by the two variables, VV and A, occurring in the prefixes.

Notice that the optionality of the second lnode depends on whether the first lnode was labeled PS:V or PS:BE. In general, contingencies expressed through prefixes result in more efficient processing since their evaluation does not involve searching and making comparisons with the chart.

The generative base component contains two rules for expanding noun—phrases. These have been collapsed to (122a):

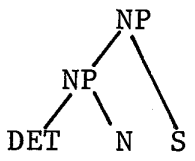
(122) (a) NP → $\left\{\begin{matrix} \text{NP} & \text{S} \\ (\text{DET}) & \text{N} \end{matrix} (\text{S})\right\}$

(b) 5, (+NP +S | +DET? +N +S?).VNP = +NP¢('VNP);

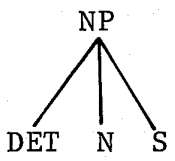
(c) 5/4, (+NP +S | +DET? +N +S?).VNP = +NP¢('VNP);

The first alternative in (122a) generates noun—phrases with relative clauses while the second generates sentential complements. This example illustrates one problem with creating a recognition grammar by simply inverting the rules of a generative grammar. (122a) can be used to generate two different structures with the same terminal string, as in (123).

(123) (a)



(b)



The MGRL rule (122b) will recover both structures, given the ambiguous underlying string, even though only one of them might be desired. The ordering restriction of (122c)

is one method of blocking the analysis (123b) since the rule cannot apply to its own output, but without additional syntactic or semantic information, there is no way of selecting the appropriate analysis for every input string.

APPENDIX: FORMAL SUMMARY OF MGRL SYNTAX

A. Macro Definitions

1. %BOOL (page 13)

(M1) %BOOL[X,AND,NEG] $\Rightarrow$ Xalternation

(M2) Xalternation $\rightarrow$ Xconjunction [| Xconjunction]^n

(M3) Xconjunction $\rightarrow$ Xitem [AND Xitem]^n

(M4) Xitem $\rightarrow$ [NEG] $\left\{ \begin{array}{c} \text{X} \\ (\ \text{Xalternation}\ ) \end{array} \right\}$

2. %SET (page 13)

(M5) %SET[X,PREFIX,MINUS,VARMOD] $\rightarrow$ Xset

(M6) Xset $\rightarrow$ Xsingleton [, Xsingleton]^n

(M7) Xsingleton $\Rightarrow$

[PREFIX] [MINUS] $\left\{ \begin{array}{l} \text{literal} \\ [\#]\ \text{valvar} \\ (\ \text{Xset}\ ) \end{array} \right\}$ [VARMOD^n]

B. Definition of the Object-Language

(D1) (page 16)

$$\text{character} \rightarrow \left\{ \begin{array}{l} \text{letter} \\ \text{digit} \end{array} \right\}$$

(D2) (page 16)

variable → character^{n}

(D3) (page 16)

literal → character^{n}

(D4) (page 16)

number → digit [digit [digit]]

(D5) (page 16)

litlist — literal [, literal]^{n}

(D6) (page 16)

declaration → number (litlist) [(litlist)]

(D7) (page 20)

grammar → [declaration^{n} ;] rule^{n}

(D8) (page 20)

rule → [D] id , rule body ;

(D9) (page 20)

id → number [/ [number] [/ number]]

(D10) (page 20)

rule body → left side = [right side]

(D11) (page 22)

$$\underline{\text{left side}} \rightarrow \left\{ [\underline{\text{lhs}}] \text{ " } \overset{\underline{\text{lhs}}}{\underline{\text{lhs}}} \text{ " } [\underline{\text{lhs}}] \right\}$$

(D12) (page 22)

$$\underline{\text{right side}} \rightarrow \underline{\text{rhs}} \text{ [" } \underline{\text{transformation}}^{n}]$$

(D13) (page 24)

$$\underline{\text{lhs}} \rightarrow \%\text{BOOL}[\underline{\text{lterm}}, \emptyset, \neg]$$

(D14) (page 24)

$$\underline{\text{rhs}} \rightarrow \%\text{BOOL}[\underline{\text{rterm}}, \emptyset, \emptyset]$$

(D15) (page 25)

$$\underline{\text{lterm}} \rightarrow [\underline{\text{prefix}}] \; [\neg] \left\{ \begin{array}{l} \overset{\underline{\text{lnode}}}{[\{\overset{\#}{@}\}]} \; (\underline{\text{lhs}}) \\ [\not\in] \; \underline{\text{valvar}} \end{array} \right\} [\underline{\text{lmod}}^{n}]$$

(D16) (page 25)

$$\underline{\text{rterm}} \rightarrow [\underline{\text{prefix}}] \left\{ \begin{array}{l} \underline{\text{rnode}} \\ (\underline{\text{rhs}}) \\ [\not\in] \; \underline{\text{valvar}} \end{array} \right\} [\underline{\text{rmod}}^{n}]$$

(D17) (page 25)

$$\underline{\text{lmod}} \rightarrow [\underline{\text{prefix}}] \left\{ \begin{array}{l} * \\ ? \\ \underline{\text{varass}} \\ (\underline{\text{lmod}}^{*}) \end{array} \right\}$$

(D18) (page 25)

$$\underline{\text{rmod}} \rightarrow [\underline{\text{prefix}}] \left\{ \begin{array}{l} ? \\ \underline{\text{varass}} \\ (\ \underline{\text{rmod}}^{n}\ ) \end{array} \right\}$$

(D19') (page 44)

$$\underline{\text{lnode}} \rightarrow \underline{\text{lspecabr}}\ [\underline{\text{lsub}}]$$

(D20') (page 44)

$$\underline{\text{rnode}} \rightarrow \underline{\text{rspecabr}}\ [\underline{\text{rsub}}]$$

(D21) (page 31)

$$\underline{\text{lspec}} \rightarrow <\ [\underline{\text{lbool}}]\ >$$

(D21a) (page 44)

$$\underline{\text{lspecabr}} \rightarrow \left\{ \begin{array}{l} [+]\ \underline{\text{lspec}} \\ +\ \%\text{SET}[\underline{\text{lpair}},\underline{\text{prefix}},-,\underline{\text{varmod}}]\ [\underline{\text{lspec}}] \end{array} \right\}$$

(D22) (page 31)

$$\underline{\text{rspec}} \rightarrow <\ [\underline{\text{rbool}}]\ >$$

(D22a) (page 44)

$$\underline{\text{rspecabr}} \rightarrow \left\{ \begin{array}{l} [+]\ \underline{\text{rspec}} \\ +\ \%\text{SET}[\underline{\text{rpair}},\underline{\text{prefix}},\emptyset,\underline{\text{varmod}}]\ [\underline{\text{rspec}}] \end{array} \right\}$$

(D23) (page 31)

$$\underline{\text{lbool}} \rightarrow \%\text{BOOL}[\underline{\text{lelement}},\_,\neg]$$

(D24) (page 31)

$$\underline{\text{rbool}} \rightarrow \%\text{BOOL}[\underline{\text{relement}},\_,\neg]$$

(D25) (page 31)

lsub → { $ / ¢ } (lhs)

(D26) (page 31)

rsub → ¢ (rhs)

(D27) (page 37)

lelement → [prefix] [¬] { lpair / { valvar / (lpair) } [varmod^n] }

(D28) (page 37)

relement → [prefix] [¬] { rpair / { valvar / (rpair) } [varmod^n] }

(D29) (page 38)

lpair → literal { : / ! } { lspec [varmod^n] / %SET[lpair,prefix,-,varmod] }

(D30) (page 38)

rpair → literal : { rspec [varmod^n] / %SET[rpair,prefix,∅,varmod] }

(D31) (page 49)

varmod → [prefix] { varass / (varmod^n) }

(D32) (page 49)

varass — . variable

(D33) (page 49)

valvar → ' variable

(D34) (page 55)

prefix → / (%BOOL[condition,&,¬])

(D35) (page 55)

$$\text{condition} \rightarrow \left\{ \begin{array}{l} \text{variable} \\ \text{valvar} = \%\text{SET}[\text{prefix},\emptyset,\emptyset,\emptyset] \end{array} \right\}$$

(D36) (page 57)

$$\text{transformation} \rightarrow \text{tname} \ (\ \text{variable} \left\{ \begin{array}{l} [,\ \text{variable}]^n \\ [,\ \text{rterm}]^n \end{array} \right\})$$

(D37) (page 57)

$$\text{tname} \rightarrow \text{character}^n$$