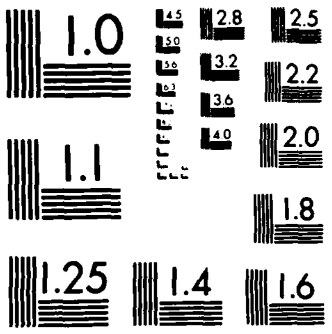
AD-A171 841   PROTEUS PARSER REFERENCE MANUAL(U) NEW YORK UNIV NY   1/1
COURANT INST OF MATHEMATICAL SCIENCES R GRISHMAN
JUL 86 N00014-85-K-0163

UNCLASSIFIED   F/G 9/2   NL


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

PROTEUS Parser Reference Manual

Ralph Grishman

PROTEUS Project Memorandum #4
July 1986

PROTEUS PROJECT

[illegible]

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University

4

PROTEUS Parser Reference Manual

Ralph Grishman

PROTEUS Project Memorandum #4

July 1986

Development of this program was supported in part by the Defense Advanced Research Projects Agency under contract N00014-85-K-0163, by the Naval Research Laboratory under contract N00014-85-K-2028, and the National Science Foundation under grant DCR-85-01843.

Table of Contents

CHAPTER 1

Introduction

1. Scope of this volume

This manual describes the English language syntactic analyzer developed by the PROTEUS Project at New York University, and the version of Restriction Language which is used to write grammars for this analyzer. This manual describes the version of the system implemented on DEC VAXen under bsd 4.2 UNIX (tm).

2. Roots

This system is a direct descendant of the Linguistic String Parser, developed by the Linguistic String Project at New York University (since 1973 in collaboration with the Computer Science Department). In particular, we have tried to maintain as much commonality as possible in the Restriction Language used for stating grammars. In developing our new implementation, we have had three objectives:

- use LISP. The current Linguistic String Parser is implemented in FORTRAN. It is therefore quite efficient but is hard to interface to AI applications, which are usually best developed in LISP. The PROTEUS system has been entirely implemented in LISP.
- remain small and modular. The Linguistic String Parser gradually became so large and complex that further modification was difficult. Through redesign and the elimination of some features, we have sought to return to a simpler, more easily modifiable system.
- accomodate different analysis algorithms. One aspect of our current research is the study of alternative analysis strategies. We have therefore tried to develop a system which could accomodate different analysis algorithms. In particular, we have designed the grammar formalism to work with both top-down and bottom-up analyzers.

3. Structure of this volume

This volume has three parts: a description of the commands for involving the English analyzer and the Restriction Language compiler; a description of Restriction Language; and a brief description of the translation rules used in compiling Restriction Language.

CHAPTER 2

Parser, Compiler, and Preprocessor

The principal program in our system is the English language syntactic analyzer, which will usually be referred to simply as *the Parser*. The Parser takes three inputs: a grammar of English, a word dictionary, and a set of English sentences. The grammar is written by the user in PROTEUS Restriction Language. Before this grammar can be used by the Parser, it must be translated into LISP. This translation is performed by the *Restriction Language Compiler* (henceforth called simply *the Compiler*).

The Compiler is controlled by a set of translation rules called the *Restriction Language Syntax*, or *RLS*. The RLS is prepared in a form described in chapter 00 of this volume. Before it can be used by the Compiler it too must be translated into LISP. This translation is done by the *RLS Preprocessor* (the Preprocessor is not driven ' y a file of rules, so we do not have an infinite regress of metacompilers). The RLS Preprocessor need be executed only when the definition of Restriction Language is modified or extended. Consequently, the Preprocessor will not be invoked by most users of the PROTEUS system.

1. Accessing the programs

In order to access these programs, the search path must be modified to include the appropriate PROTEUS directories. This is normally done by a command in the ".login" file. For ACF5 at NYU, the command is

```
source ~proteus/usersetup
```

2. Invoking the Parser

The Parser reads two input files: a *grammar* and a *dictionary*. The Parser is invoked with the command

xparse -g *grammar* -d *dictionary*

If the arguments are omitted on the command line, they will be prompted for interactively. If the dictionary is incorporated in the grammar file -- so that there is no separate dictionary file -- the value **none** should be given for *dictionary*.

When the Parser begins executing, it produces the prompt **sentence>**. In response one can either type a sentence to be parsed or a Parser command; these commands are described in the section just below. A sentence is a series of words, separated by one or more spaces, and ending in a period or question-mark. A sentence may extend over several lines. Case is ignored on sentence input: all upper-cas letters are converted to lower-case before the words are looked up in the dictionary.

As soon as a sentence is entered, it will be parsed. If parses are obtained, then (under the default settings of the Parser's switches), the parses and regularized parses will be printed.

2.1. Parser commands

Operation of the Parser in controlled by Parser commands. These commands consist of an asterisk (*), a command name, and (for some commands) an argument to the command. Some commands turn on and off switches which control future parsing operations; other comnmands cause immediate actions. The commands are as follows:

2.1.1. *exit

***exit** terminates a run of the parser.

2.1.2. *lisp

***lisp** turns control over to the LISP interpreter. To resume parsing, invoke the function **parser-top-level.**

2.1.3. *[no]trees

***trees** causes each parse that is obtained from the analysis of subsequent sentences to be printed in a parenthesized, indented form. ***notrees** suppresses the printing of parses. Default is ***trees.**

2.1.4. *[no]empties

***empties** causes subsequent parse trees to be printed with all nodes of the tree included; ***noempties** causes subsequent trees to be printed with empty nodes (nodes not subsuming any sentence words) omitted. Default is ***empties.**

2.1.5. *[no]attributes

***attributes** causes subsequent parse trees to be printed with the attributes assigned to a node (except for the translation attribute "Xn") listed after each node name; ***noattributes** suppresses the printing of this information. The default is ***noattributes.**

2.1.6. *draw

The command ***draw** *n*, when entered after a sentence has been parsed, causes the *n*th parse tree of the sentence to be drawn on the screen. The tree-drawing program provides several commands for manipulating the tree. In particular, if the entire tree does not fit on the screen, commands are provided to shift the portion visible on the screen. To list the commands available in the tree-drawing program, type "help". To return to the parser, type "quit".

2.1.7. *[no]failtrace

The ***failtrace** command causes a trace message to be printed whenever a restriction fails during the parsing of subsequent sentences. The message gives the name of the restriction, the node at which the restriction was housed, and the words subsumed by that node. ***nofailtrace** turns off this trace. The default it ***nofailtrace**.

3. Invoking the Compiler

The grammars used for analyzing natural language are written in PROTEUS Restriction Language, which is described in Chapter 3. Before it can be used by the Parser, such a grammar must be translated into Lisp. This translation is performed by the Compiler, which is invoked with the command

```
xcompile -g Restriction-Language-file -o Lisp-file [-r RLS-file]
```

The Compiler reads the grammar on the *Restriction-Language-file*, translates it into Lisp, and writes it on the *Lisp-file*. If these arguments are omitted in the command line, they will be prompted for interactively. The RLS (the translation rules used in compiling) are normally taken from file "rls.out" in the PROTEUS system directory, but this choice can be overridden by specifying an explicit *RLS-file*.

Since the generated file is a Lisp program, it can be sent through the Lisp compiler (liszt) to produce a machine language file. The increase in parsing speed which can be obtained in this way varies with the complexity of the restrictions; typically, speed-up factors of 1.5 to 2 can be expected.

4. Invoking the Preprocessor

The Preprocessor translates the RLS (Restriction Language Syntax) into the form needed by the Compiler. It is invoked by the command

```
xpreproc
```

The Preprocessor prompts interactively for the name of the source RLS file and the name of the file to be generated.

CHAPTER 3

Preparing a grammar: BNF and Lists

1. The structure of the grammar

An English grammar consists of four components: BNF; lists; routines and restrictions; and word dictionary. These components must appear in this order. The BNF and lists are described in this chapter; the routines and restrictions are described in the next chapter, and the form of the word dictionary is considered in chapter 5.

The grammar defines many different types of symbols, such as nonterminal symbols, terminal symbols, routine names, attributes, etc. Any symbol name valid in LISP may be used: in particular, any sequence of upper and lower case letters, digits, hyphens, and underscores is allowed, provided the name contains at least one non-digit. There is one exception: names beginning with "X" are reserved as register names.

2. BNF

The first component of the grammar, the BNF, specifies the context - free portion of the grammar in an extension of Backus Normal Form. Most of the definitions adhere strictly to BNF form, with double quotes (") used to enclose any constants in the definition:

```
<QUACK> ::= <A> <B> | "OINK".
```

(like all statements, BNF statements must end in a period.) Each alternative is called an option, and each item in an option is called an element. Constants, such as "OINK" above, are also called literals. Terminal symbols (grammatical categories) are indicated by placing an * to the left of their name:

```
<NVAR> ::= <*N> | <*VING>.
```

Any terminal symbol beginning with the letters NULL designates the empty string (this allows for symbols like NULLN, NULLOBJ, and NULLC to indicate empty strings of different linguistic import).

The BNF section has two functions: one, to specify the context-free grammar, the other to declare certain names as nonterminal or terminal symbols, i.e., valid node names. The subsequent sections of the grammar (lists, restrictions, word dictionary) will not accept a symbol as a node name unless it has appeared somewhere in the BNF.

2.1. Parentheticals

The element/option structure of BNF may be nested using parentheses:

```
<RNWH> ::= ( "WHO" | "WHICH" ) <VERB> <OBJECT>.
```

Such parenthesized structures are expanded as part of the process of compiling a grammar, so the definition above is precisely equivalent to

```
<RNWH> ::= "WHO" <VERB> <OBJECT> |
        "WHICH" <VERB> <OBJECT>.
```

2.2. Translation rules

Each option is optionally followed by a colon (:) and a lisp expression. This expression is the translation rule for this option, and is used in generating the regularized parse tree. Further explanation of these translation rules appears in Chapter 00.

3. Lists

There are two kinds of lists: type lists and attribute lists. They may appear in this section in any order.

3.1. Type lists

There are certain sets among the BNF symbols, such as the adjunct strings, which are frequently referred to in the restrictions. Rather than enumerate these sets each time, they may be given a name, and this name used in the restrictions in place of the set. These sets are called *types*, and their declarations *type lists*. A type list declaration consists of the word TYPE, followed by the name of the list, an '=', and then a series of BNF symbols separated by commas. For example, a declaration for type RADJSET (Right ADJunct SET) might be

```
TYPE RADJSET = RN, RW, RQ, RV, RA, RA1, RD.
```

sets may overlap--a symbol may belong to any number of types.

There are several type lists which have special significance to the syntactic regularization mechanism and the conjunction mechanism. These are described in the later chapters covering these mechanisms.

3.2. Attribute lists

Every attribute referenced in the restrictions and word dictionary must be declared in an attribute list. This list has the form

```
ATTRIBUTE = SINGULAR, PLURAL, COMPARATIVE.
```

CHAPTER 4

Preparing a Grammar: Restrictions

This chapter describes the Restriction Language which is used to specify the procedural portion of the grammar: the restrictions and the grammar routines. This chapter is not intended as an introduction to the Restriction Language; a separate publication, *An Introduction to Restriction Language*, has been prepared for that purpose. Readers may also wish to consult Sager and Grishman, "The Restriction Language for Computer Grammars of Natural Language," *Comm. Assn. Computing Machinery 18*, 390 (1975) for an overview of the earlier version of the language, as used in the NYU Linguistic String Parser

A more formal definition of the language is also available to the reader: the RLS (Restriction Language Syntax). As explained in chapter 6, the RLS is used to translate Restriction Language into LISP. The RLS and the Restriction Language run-time library (a set of LISP functions on file egruntime.l), taken together, provide a more precise but more inscrutable exposition of Restriction Language than that given below.

1. Basic Concepts

The procedural portion of a grammar contains a set of restrictions. These restrictions are invoked by the context-free parser to verify the correctness of a partial sentence analysis. A restriction may either *succeed*, which indicates that the partial analysis is acceptable (with respect to a particular gramatical constraint), or *fail*, which indicates that the analysis is unacceptable and a different analysis should be tried.

1.1. Data structures

Restrictions are able to examine three different data structures: the parse tree, attribute lists, and sentence words. The parse tree consists of non-terminal and terminal *nodes*. Associated with each non-NULL terminal node is the word it has matched in the sentence; this word can be accessed (starting from the terminal node) by the **WORD** routine in Restriction Language.

Associated with every node in the tree is an attribute list. Each attribute list consists of a set of attribute names (as declared in the **ATTRIBUTE** list of the grammar) and associated values. This is analogous to (and is represented internally as) a LISP property list. The "associated value" can be *true*, another attribute list (thus creating a tree of attributes), or any other list structure. The attribute list for a terminal node is initially obtained from the word definition of the word matched by that node; the attribute lists of other nodes are initially empty. Attributes can be added to and removed from a node by the **ASSIGN ATTRIBUTE** and **ERASE ATTRIBUTE** commands of Restriction Language. The attribute list of a node can be accessed by the **ATTRIBUTE** routine and by the **HAS ATTRIBUTE** predicate.

For historical reasons the variables of Restriction Language are called *registers*. Register names are distinguished by the initial leter **X**: Any symbol name beginning with an X may be used as a register name. A register can point to any one of the three types of data structure: a node of the parse tree, a list element, or a sentence word. Registers may be declared as the formal parameters and local variables of grammar routines. Such variables are created when a routine is involved and destroyed on routine exit. Undeclared registers are global in scope and retain their value until reassigned.

In describing the semantics of Restriction Language, we will often refer to the notion of a *current position:* at each step in the execution of a restriction, we say that the restriction is "looking at" some parse tree node, some list element, or some sentence word. In the LISP implementation, this "current position" corresponds in some cases to the value bound to the variable *here*, and in some cases to the values returned by grammar routines and other functions.

Every Restriction Language operation also returns an indication of *success* or *failure*. This information is used by the logical operators (AND, OR, etc.) to control the flow of execution as well as determine the success/failure of the logical operation. Ultimately this process determines the success or failure of the entire restriction. In the LISP implementation, failure is indicated by returning a value of *nil;* it is therefore not possible to have a routine which succeeds and returns nil.

1.2. Classes of Symbols

The symbols in the grammar are classified into various *classes*. The BNF component has *non-terminal* and *terminal* symbols. The BNF component also has *literals* (represented by strings enclosed in double quotes ("...")). Non-terminal and terminal symbols and literals are all possible *node names*. The lists component introduces *attributes* and *type list names*. The restriction component introduces several additional types of symbols, including *register names*.

In describing below the Restriction Language, we have included BNF descriptions of some portions of the language. This BNF shall use the following terminal symbols:

<*terminal>	a terminal symbol from the BNF component
<*non-terminal>	a non-terminal symbol from the BNF component
<node>	a node name: a non-terminal, terminal, or literal
<*attribute>	an attribute
<*type>	a type list name
<*register>	a register name

1.3. Metagrammatical notation

In describing the Restriction Language, we have supplemented our narrative with BNF specifications of some of the constricts. A complete BNF specification of the language is contained within the RLS. The BNF used is the same as that used in the context-free component of the English grammar, with two additions: the notation

(x)*

means "zero or more instances of x", and the notation

[x]

means "zero or one instances of x". Readers comparing the BNF given in this chapter with the RLS will observe a few changes in definitions and symbol names, intended to make the BNF easier to read (some RLS definitions are constructed to simplify the semantics or speed the parsing).

One special note: the articles A, AN, and THE may be used freely in Restriction Language to improve readability. They are ignored by the compiler and hence do not appear in the language description given below.

2. Types of Statements

The language has three types of statement: restrictions, grammar routines, and substatements. These statements differ in the way they are invoked: restrictions are invoked directly by the parser. The routines and substatements both serve as internal procedures for the restrictions; the substatements are simple parameterless procedures which can succeed or fail but do not return values, while routines provide parameters and local variables and return values. (Routines and substatements were included in the language to meet different needs: substatements were provided to divide and organize the tests performed by complex restrictions, while routines were intended primarily for the procedures which locate parse tree nodes corresponding to basic grammatical relations.)

The bodies of restrictions and substatements have the same structure; they differ only in their headers, corresponding to their different means of invocation. Routine bodies, which are described in a separate section later in this chapter, are similar to the subject portion of declarative restriction statements.

3. Housing

Restrictions have the form

```
<*name> "=" <housing> ":" <body> "."
```

where *name*, the name assigned to the restriction, is any valid symbol name. This name is not significant internally within the grammar, but is used to refer to the restriction in traces and other output.

The *housing* specifies when a restriction is to be executed. It has the syntax

```
<housing>      ::=  <indef> [ <after> ]
<indef>        ::=  "IN" <def-or-type> ("," <def-or-type>)*
<def-or-type>  ::=  <*non-terminal> | <*type>
<after>        ::=     "AFTER"
                     ("OPTION" <opt-spec> |
                      "ELEMENT" <*integer> <of-option> |
                      "EVERY" "ELEMENT" <of-option> |
                      <node> <of-option> )
<of-option>    ::=  "OF" "OPTION" <opt-spec> | <*null>
<opt-spec>     ::=  <*integer> | <node>
```

The housing has two parts: the first (*indef*) specifies the definitions to which the restriction applies; the second (*after*) specifies the point in the elaboration of the definition when the restriction should be executed.

indef is a list which can contain both definition names and type list names. A definition name (*non-terminal*) indicates that the restriction is to be housed in the specified definition. A type list name (*type*) indicates that the restriction is to be housed in each definition belonging to the type list.

The *after* construct specifies to which options of the definition(s) the restriction applies, and at what point in the construction of a parse subtree corresponding to the definition the restriction is to be executed. These subtrees are built incrementally; if a definition *D* has several elements, daughter nodes will be added below *D* one by one as the elements are matched. We can specify, through the housing, that a restriction is to be executed after one element, two elements, or all the elements of *D* have been completed.

The *after* construct may take four forms. The first form is

AFTER OPTION <opt-spec>

which specifies that the restriction is to be executed after all elements of the specified option have been completed (in other words, it is housed on the last element of that option). The option may be specified either by the name of the first element of the option, or by the number of the option. For example, if we have the definition

<letter> ::= <greeting> <body> | <insult> <complaint>

the following two housings are equivalent:

IN letter AFTER OPTION insult:
IN letter AFTER OPTION 2:

The second form,

AFTER ELEMENT <*integer> [OF OPTION <opt-spec>]

specifies both the option to which the restriction applies and the point in the completion of that option that the restriction is to be executed. For example, with the above definition of *letter*, the housing

AFTER ELEMENT 1 OF OPTION insult:

would execute the restriction after element *insult* had been matched, but before element *complaint* had been matched. If the **OF OPTION** phrase is omitted, the restriction is applied *to all* options of the definition.

The third form,

AFTER <node> [OF OPTION <opt-spec>]

provides the same capabilities as the second, but allows the element to be specified by name rather than by number. Again, if the **OF OPTION** phrase is omitted, the restriction is applied to all the options.

The final form,

AFTER EVERY ELEMENT [OF OPTION <opt-spec>]

allows the restriction to be executed repeatedly, once after each element of the definition has been completed. The *after* construct is optional. If it is omitted, the restriction will be housed on the last element of every option of the specified definitions.

4. Logical connectives

The *body* of a restriction or substatement is a *compound* statement, which is built up out of *simple* sentences using a variety of connectives according to the following syntax:

```
<compound> ::=    "IN" <subject> "," <compound> |
                  "BOTH" <compound> "AND" <compound> |
                  "EITHER" <compound> "OR" <compound> |
                  "IF" <compound> "THEN" <compound>
                           [ "ELSE" <compound> ] |
                  "NOT" <compound> |
                  <simple>.
```

The **IN** phrase resets the present position at which the following sentence or sentences will be evaluated. For example,

```
IN ASSERTION,
  BOTH CORE OF SUBJECT IS PLURAL
  AND CORE OF VERB IS PLURAL
```

causes the two sentences, *CORE OF SUBJECT IS PLURAL* and *CORE OF VERB IS PLURAL* to be evaluated while looking at the node *ASSERTION*. The allowed form for *subject* are described in a later section.

The other connectives provide the basic Boolean operations: conjunction, disjunction, implication, and negation. In all cases the language evaluates only as many consitutent sentences as are needed to determine the success or failure of the compound sentence. Specifically, in the case of **BOTH** *A* **AND** *B*, if *A* fails, *B* is not executed. In the case of **EITHER** *A* **OR** *B*, if *A* succeeds, *B* is not executed.

The rule for the Boolean operators is that each operand is evaluated with the same initial position - the current position when the Boolean operator is invoked. This means, for example, that if

```
BOTH $A AND $B
```

is executed while positioned at parse node V, both $A and $B will be executed with initial position = parse node V.

5. Elementary sentence types

The sentences in Restriction Language are of two types: declarative sentences and assignment operations. Declarative sentences have a subject-predicate structure (declarative sentences state conditions; the sentence succeeds if the condition is true). Assignment operations can assign a value either to a register or to a node attribute.

6. Declarative sentences: subject forms

The primary sentence form in Restriction Language is a declarative sentence with a subject-predicate structure. Roughly speaking, the subject locates a parse tree node, a list, or a sentence word and the predicate asserts some property of the located item, although this division is not strictly observed. The sentence will fail if either the item cannot be located (subject failure) or the property does not hold (predicate failure).

Locating an item of interest may require a series of actions. If subj-a and subj-b are subject forms, then

```
subj-a OF subj-b
```

means to apply the locating action of subj-b followed by that of subj-a. For example,

```
CORE OF COELEMENT SUBJECT
```

applies the actions of COELEMENT SUBJECT followed by those of CORE. The OF construct is only allowed after some subject forms, as indicated in the detailed syntax given below.

A register name may appear after some subject forms, to indicate that the item located by the subject is to be stored in the specified register. Thus

```
CORE X1
```

means locate the CORE and store it in register X1. In a "subj-a OF subj-b" construction, a register may follow each subject form:

```
CORE X1 OF COELEMENT SUBJECT X2
```

means to locate COELEMENT SUBJECT, store that node in X2, locate CORE, and store that node in X1.

6.1. Subjects which invoke routines

The primary subject form is a reference to a routine. It has the form

```
<*routine-name> <args> [<*register>] ["OF" <subject>]
```

where

```
<args>::= <node>|
          "(" <node> ("," <node>)* ")" |
          <*null>
```

This form invokes routine *routine-name* with argument(s) *args*. (These are in addition to the implicit argument, namely the "current position" when the routine is invoked.) The routine may be either one predefined by the system or one defined in the grammar. Four routines are predefined in the current system. Three of these take no explicit arguments:

NAME
(if looking at a node of the parse tree) returns the name of that node

WORD
(if looking at a terminal node of the parse tree) the word matching that node

LAST-ELEMENT
(if looking at a non-terminal node of the parse tree) the last immediate descendant of that node

The fourth routine has one explicit argument:

ATTRIBUTE(*a*)
(if looking at a node of the parse tree or at an attribute list) if the node or list includes an attribute *a*, succeed and return "looking at" the value associated with that attribute.

A node name appearing by itself:

```
<node>[<*register>]["OF"<subject>]
```

causes the grammar routine STARTAT to be invoked with that node name as argument. The normal definition of STARTAT has the following effect: if the current node has the name specified, the routine remains there; otherwise the level immediately below the current node is searched, left to right, for a node of that name.

6.2. FIRST-ELEMENT

The FIRST-ELEMENT subject has the form

```
FIRST-ELEMENT [WHICH <predicate>] [OF <subject>]
```

Starting at a node of the parse tree, it searches the level below that node from left to right for a node satisfying *predicate* (the allowable predicate forms are described below). If not initially at a node, or if no immediate descendant satisfies *predicate*, the subject fails. Examples of this construct are

FIRST-ELEMENT WHICH IS NOT EMPTY
FIRST-ELEMENT WHICH HAS ATTRIBUTE LOVEABLE

Omitting the predicate is equivalent to having a predicate of *true* -- the subject goes to the leftmost immediate descendant of the current node.

FIRST-ELEMENT is intended for use primarily in grammar routines, as the basic construct for searching downward in the parse tree.

6.3. Other subject forms

Two other simple subject forms are allowed A register name by itself causes the restriction to "look at" the contents of the register. The subject form "PRESENT-NODE' causes the restriction to stay wherever it is (at the node housing the restriction, if there were no preceding "IN <subject>" phrases).

7. Declarative sentences: predicate forms

The predicate is the second part of the subject-predicate declarative sentence form. The predicate assorts some property of the item located by the subject; if the property is true, the predicate succeeds. In addition to the predicates to be described below, there is a dummy predicate "EXISTS";

<subject> EXISTS

will succeed precisely if <*subject*> succeeds.

An (affirmative) subject-predicate construct will succeed if the subject succeeds and the predicate succeeds. For each affirmative predicate there is a corresponding negative predicate, formed by inserting "NOT" according to the usual rules for English:

IS - IS NOT
HAS - DOES NOT HAVE
EQUALS - DOES NOT EQUAL

A sentence with a negative predicate will succeed if the corresponding affirmative sentence would fail (either because the subject fails or the predicate fails)[1]. There is no negative form for the EXISTS predicate.

7.1. Node name tests

The predicate

IS NAMED <node>

succeeds if the current position is a parse tree node with the name specified. This predicate may be stated more tersely as

IS <node>

The name may also be given by a register:

IS NAMED <*register>

succeeds if *register* contains a symbol equal to the name of the current node (the word "NAMED" may *not* be omitted in this case, when the predicate references a register). The test

[1] This is different from the interpretation of negative sentences in the original LSP Restriction Language.

IS OF TYPE <*type>

succeeds if the name of the current node is on the type list **type*.

7.2. Attribute tests

To each node in the parse tree we can assign a set of attributes. Terminal nodes receive attributes from the dictionary entries for the words they match. In addition, any node can be assigned a node attribute by the ASSIGN NODE ATTRIBUTE command, which is described below. Each node attribute has a value; this value can either be *true* or a set of attributes and values.

The predicate

HAS ATTRIBUTE <*attribute>

is used to test for the presence of a particular attribute on a node or on an attribute list. This predicate may be stated more tersely as

IS <*attribute>

The attribute name may also be given by a register:

HAS ATTRIBUTE <*register>

succeeds if the current position (node or attribute list) has an attribute equal to the symbol in *register* (again, the terser form "IS" cannot be used when the predicate regerences a register).

Word definitions are organized hierarchically, with the values of some of the attributes being themselves attribute lists. If we wanted to test whether a node we have located is a TV with attribute OBJLIST and sub-attribute NSTGO (i.e, N: (OBJLIST: (NSTGO))) we would have to write several sentences using the predicates just described; for example,

```
BOTH PRESENT-NODE IS TV
AND ATTRIBUTE OBJLIST OF PRESENT-NODE
HAS ATTRIBUTE NSTGO
```

Restriction Language allows us to combine the two attribute tests in a single predicate, using the operator ":" (meaning "with attribute"):

```
BOTH PRESENT-NODE IS TV
AND PRESENT-NODE HAS ATTRIBUTE OBJLIST:NSTGO
```

The general form of this predicate is

HAS ATTRIBUTE <*attribute> (":" (<symbol> | <*register>))*

or

IS <*attribute> (":" (<symbol> | <*register>))*

Taking this one step further, we may combine the tests for node name and attributes:

PRESENT-NODE IS TV:OBJLIST:NSTGO

corresponding to the general predicate form

IS <node> (":" (<symbol> | <*register>))*

7.3. Other node tests

Two other predicates are provided for testing parse tree nodes.

IS TERMINAL

succeeds if the node is a terminal node of the tree;

IS EMPTY

succeeds if the node subsumes no sentence words (i.e., if all the terminal nodes below this node are NULL nodes).

7.4. Equality test

Several subject forms (such as the routines NAME and WORD) evaluate to LISP symbols. To test these symbols, Restriction Language provides the predicate.

EQUALS (<symbol> | <*register>)

which succeeds if the value of the subject is equal to the symbol or the contents of the register.

8. Assignment statements

In addition to the declarative statement forms described in the last two sections, Restriction language provides commands for assigning values to registers and assigning attributes to nodes.

8.1. Register assignment

The statement form

<*register> "=" <subject>

assigns to *register* the value of *subject*. This value may be a node, an attribute list, or a symbol. For example,

```
X-SUBJ=CORE OF SUBJECT
X-OLIST=ATTRIBUTE OBJLIST OF PRESENT-NODE
```

8.2. Node attribute assignment

The statement form

ASSIGN ATTRIBUTE <*attribute> [WITH VALUE (<symbol> | <*register>)]

assigns to the node at which the restriction is housed the specified attribute and value. If the VALUE phrase is omitted, the attribute will be given the value *true*. The statement form

ERASE ATTRIBUTE <*attribute>

erases (deletes) the specified attribute from the node at which the restriction is housed.

9. Substatements

Substatements have the form

<*substatement-name> " " <body> "."

The *substatement name* must begin with a "$", and otherwise conform to the rules for symbol names. The *body* of the substatement, just like the body of a restriction, may be any *compound* statement.

Substatements act as parameterless procedures, which are evaluated only to determine whether they succeed or fail. They are referenced by writing the name of the substatement; this may appear at any point where a compound statement may appear.

For example,

```
CHECK-PETS = IN PETS: BOTH $CATS AND $DOGS.
 $CATS = ELEMENT CATS IS NOT EMPTY.
 $DOGS = ELEMENT DOGS IS NOT EMPTY.
```

is equivalent to

```
CHECK-PETS = IN PETS:
 BOTH ELEMENT CATS IS NOT EMPTY
 AND ELEMENT DOGS IS NOT EMPTY.
```

A substatement is considered to "belong to" the immediately preceeding restriction or routine. It can be referenced only by that restriction/routine or by other substatements which belong to that restriction/routine. The same name can be used to label different substatements within different restrictions or routines.

10. Routines

Routines have the form

```
ROUTINE <*name> "=" ["(" <formals> ")"] <compound-subj> "."
```

where ***name*** is the name of the routine (any valid symbol name), and *formals* -- the list of formal parameters -- is a list of one or more register names separated by commas.

The semantics of routine parameters is modeled after LISP. Name scoping for registers is dynamic (undeclared registers are globally accessible). Parameters are passed by value. The number of arguments in a call must exactly match the number of formal parameters.

Routine names are global: a routine may be referenced in any other routine or restriction. Nested and recursive calls are allowed.

In constrast to substatements, routines are evaluated for their value (in addition to their success or failure). The body of a routine may be any valid *subject* form, as described above. In addition, certain *compound subjects* are allowed in routines. They have the syntax:

```
<compound-subj>      ::=
        IF <compound> THEN <compound-subj> ELSE <compound-subj>
        EITHER <compound-subj> OR <compound-subj> |
        <subject>.
```

The **IF ... THEN ... ELSE ...** construct has the following semantics: the statement following the **IF** is evaluated; if it succeeds, the compound subject following the **THEN** is evaluated and returned as the value of the entire expression; otherwise the compound subject following the **ELSE** is evaluated and returned as the value of the expression. The routine STARTAT contains an example of such a construct:

```
ROUTINE STARTAT (X) =
 IF PRESENT-NODE IS NAMED X
   THEN PRESENT-NODE
   ELSE FIRST-ELEMENT WHICH IS NAMED X.
```

The **EITHER ... OR ...** compound subject has the following semantics: the first *compound-subj* is evaluated; if it succeeds, its value is returned as the value of the entire expression. Otherwise, the second *compound-subj* is evaluated and returned as the value of the entire expression.

CHAPTER 5

Preparing a Word Dictionary

The word dictionary must contain the definitions for all words (and punctuation) in the text to be parsed. Each word definition specifies a set of categories and attributes for a word. Conventionally, the words are organized in alphabetical order; the Parser, however, does not require any particular ordering of entries.

The dictionary may be provided in two ways: either as a part of the grammar file, or as a separate file. The format of the dictionary entries in the two cases is quite different. For dictionaries prepared as part of the grammar file, the format is a minimal subset of the format accepted by the Linguistic String Parser. This simple format is suitable for small teaching grammars. When prepared as a separate file, the dictionary entries take the form of cals on LISP macros. Those macros automatically generate definitions of inflected forms of verbs and nouns. In addition, this form of dictionary entry does not have to be processed by the Compiler. These features make the separate dictionary file more suitable for larger applications. The dictionary-with-grammar format is described in the next section; The separate-dictionary format in the section following.

The word dictionary entries shown here are examples of form only, and are frequently incomplete or linguistically incorrect.

1. Dictionary with Grammar

If the dictionary is incorporated in the grammar file, it must follow the BNF and ATTRIBUTE list; conventionally, the dictionary goes at the end of the grammar file.

A word definition has the structure of a tree. At the root of the tree is the word itself. Each immediate descendent of the root is a category (a terminal symbol of the grammar). The subtree below each category gives the attributes of a category, the attributes of the attributes, etc.

The simplest form of a word definition consists of a word followed by a list of categories, separated by commas:

```
FISH N, V, TV.
```

The attributes of a category are written after that category, separated by commas, enclosed in parentheses, and preceded by a colon:

```
BOOK N:(SINGULAR,NONHUMAN).
```

An attribute may be assigned attributes of its own in a similar fashion:

```
ENJOYS TV:(SINGULAR,OBJLIST:(NSTGO,VINGO)).
```

An attribute must be a terminal or nonterminal symbol of the BNF grammar, a symbol on the attribute list, or a quoted character string.

Blanks may be used freely, except within symbol names, to enhance readability. A definition may extend over more than one line and must be terminated by a period.

Certain words, which are referenced in the grammar only as literals, are not assigned any categories. They must nontheless appear in the word dictionary:

THAT.

Some words must be enclosed in double quotes, namely, those words containing a special character not allowed in LISP symbols, and the articles A, AN, and THE, which are otherwise considered noise words and ignored by the Restriction Language compiler. For example,

"."

"THE" T:(DEFINITE).

2. Separate Dictionary File

If the dictionary is prepared as a separate file, it takes the form of a set of calls on LISP macros. This file is used directly by the Parser; it is *not* preprocessed by the Compiler.

The basic macro for defining words is **word**. It takes two arguments:

(word *word category-attribute-list*)

word is the word itself. *category-attribute-list* is a list of the categories assigned to the word enclosed in parentheses, with each category optionally followed by an attribute list:

(category [attribute-list] category [attribute-list]...)

Each *attribute-list* is in turn a parenthesized list of attributes, with each attribute optionally followed by its value (another attribute list):

(attribute [value] attribute [value]...)

If *value* is omitted, the value associated with the attribute is *true*.

For example, the entry shown in the previous section for "enjoys",

enjoys TV: (SINGULAR, OBJLIST: (NSTGO, VINGO)).

would have the following form if it appeared on a separate dictionary file:

(word enjoys (TV (SINGULAR OBJLIST (NSTGO VINGO))))

2.1. noun and verb macros

In addition to the basic **word** macro, two macros are provided for automatically generating the inflected forms of nouns and verbs. These two macros take *keyword* arguments; in other words, the macro call is of the form

(macro-name keyword argument keyword argument...)

The keywords are distinguished by ending in a colon (":"). Most of the arguments are optional; if the keyword and associated argument are omitted, a default value will be used (as described below for each argument).

The **noun** macro takes four arguments:

root:	(required) the singular form of the noun
plural:	(optional) the plural form of the noun. If omitted, uses singular form of noun + "s".
Xn:	(optional) the value of the **Xn** (translation) attribute of the word, which is used in composing the regularized parse tree. If omitted, uses singular form of noun.

attributes:	(optional) a list of the attributes and associated values (in a form similar to *attribute-list* above) to be assigned to both the singular and plural forms of the word (in addition to the Xn attribute, and the SINGULAR and PLURAL attributes). If omitted, no additional attributes are assigned.

To illustrate the **noun** macro, consider the example

```
(noun root: cat attributes: (ncount))
```

This would have the same effect as

```
(word cat (SINGULAR NCOUNT Xn (cat singular))
(word cats (PLURAL NCOUNT Xn (cat plural))
```

The **verb** macro takes eight arguments:

root:	(required) the infinitive and third-person plural form of the verb
3psing:	(optional) the third-person singular (present tense) form of the verb. If omitted, uses *root* + "s".
past:	(optional) the past tense form of the verb. If omitted, uses root + "ed" (or root + "d" if root ends in "e").
pastpart:	(optional) the past participle of the verb. If omitted, uses past tense form.
prespart:	(optional) The present participle of the verb. If omitted, uses root - final "e", if present + "ing".
objlist:	(required) the list of acceptable objects for the verb
Xn:	(optional) the value of the *Xn* (translation) attribute of the verb (combined with "present" or "past" for tensed verb forms). This attribute is used in composing the regularized parse tree. If omitted, uses root form of verb.
ttributes:	(optional) a list of the attributes and values (in a form similar to *attribute-list* above) to be assigned to all forms of the verb (in addition to the OBJLIST and Xn attributes). If omitted, no additional attributes are assigned.

To illustrate the effect of the **verb** macro, consider the example

```
(verb root: bake objlist: (NULLOBJ NSTGO))
```

This has the same effect as

```
(word bake    (V (Xn bake OBJLIST (NULLOBJ NSTGO))
               TV (PLURAL Xn (present bake)
                          OBJLIST (NULLOBJ NSTGO))))
(word bakes   (TV (SINGULAR Xn (present bake)
                          OBJLIST (NULLOBJ NSTGO))))
(word baked   (TV (Xn (past bake) OBJLIST (NULLOBJ NSTGO))
               VEN (Xn bake OBJLIST (NULLOBJ NSTGO)
                          POBJLIST (NULLOBJ))))
(word baking  (VING (Xn (prog bake) OBJLIST (NULLOBJ NSTGO))))
```

2.2. Multiple definitions

A dictionary can contain two (or more) definitions for the same word. In such a case, the category lists of the several definitions are concatenated to form the category list used by the parser. For example, if the dictionary contained the entries

```
(noun  root: answer)
(verb  root: answer  objlist: (NULLOBJ NSTGO))
```

the word "answer" would be assigned the categories N, V, and TV, and the word "answers" would be assigned the categories N and TV.

CHAPTER 6

Translation Rules

The context-free (BNF) component of the grammar can be extended to a *translation grammar* by associating a *translation rule* with each production (each BNF option). We associate a *translation* (a LISP list structure) with each node in the parse tree of a sentence we have analyzed. The translations of the terminal nodes are obtained from the corresponding dictionary entries. The translations of the non-terminal nodes are then computed bottom-up in a compositional fashion. Associated with the production which expanded a non-terminal node is a translation rule which computes the translation of this node as a function of the translations of the nodes immediately below in the tree. The translation of the root node of the tree is the translation of the entire sentence.

These translation rules are used in the PROTEUS system to produce a regularized syntactic structure -- prinicipally, to map all types of clauses into an operator-operand form. The rules could also be used to produce something closer to a logical form, in the manner of Montague and Gazdar. What follows is a rather terse summary of the form of the translation rules. A more discursive description of the form of these rules, and how they are used, may be found in *Syntactic Regularization in PROTEUS* by Jean Mark Gawron (PROTEUS Project Memo #5).

1. Where they go

Each option in the BNF component may be followed by a translation rule; the general form of a BNF rule is

```
<symbol> ::  option : {translation-rule}  option : {translation-rule}  ....
```

If the translation rule is omitted (along with the colon and braces), a default rule of "NULLSEM" is used, which yields a translation value of *nil*.

2. Translations of terminal nodes

The translation of a terminal node is taken from the *Xn* attribute of that node. As noted above in the description of the word dictionary, both the **noun** and **verb** macros include an **Xn:** argument to specify the translation of the word.

3. Elements and structures of translation rules

Translation rules are assembled using a small set of elements and combining rules.

3.1. Elements

The basic elements of the translation rules are of two types: constants and references to translations of nodes on the level immediately below. A node on the level immediately below can be referred to by its name or by its position (counting the leftmost immediate descendant as 1). Thus the following two rules both specify that the translation of node *A* is equal to the translation of node *C* immediately below:

<A> ::= <B> <C> <D> :{C}.
<A> ::= <B> <C> <D> :{2}.

A reference by position is particularly useful when parenthesized expressions are used in the BNF. The rule

<A> ::= <B> (<HE> | <HO>) <D> : {2}.

will be expanded into a rule with two options:

<A> ::= <B> <HE> <D> : {2} | <B> <HO> <D> : {2}.

Reference by position is also needed when a BNF rule contains two elements with the same name. In such cases a reference by name gets the translation of the first (leftmost) element. Thus the following two rules are equivalent:

<A> ::= <B> <C> <D> <C> :{C}.
<A> ::= <B> <C> <D> <C> :{2}.

Any LISP atom which is not a symbol of the grammar is interpreted as a constant. For example,

<A> ::= <B> <C> <D> :{quack}.

specifies that the translation of *A* is the symbol *quack*, regardless of the translation of the immediate constituents of *A*. (A caution: the compiler does not check that a grammar symbol specified in a translation rule is an element of BNF rule with which it is associated; thus the following

<A> ::= <B> <C> <D> :{E}.

would compile, assuming *E* was a grammar symbol, but would fail (return nil as the translation) during parsing.)

3.2. List structures

These translation rule elements can be combined into list structures using standard parenthetical notation. For example, "(quack moo)" creates a list with two elements, "quack" and "moo". Each component of a parenthesized list can be either another list or one of the elements described above (a constant or a reference to an immediate constituent). If a component is preceded by an exclamation mark (!) then it will be *spliced* into the list. For example, if the translation of node *D* is "(ho hum)", then the rule

<A> ::= <B> <C> <D> : {(quack D)}.

with yield a translation for A of "(quack (ho hum))", while

<A> ::= <B> <C> <D> : {(quack ! D)}.

with yield a translation for A of "(quack ho hum)".

If the name of an element in the translation rule is followed by an asterisk (*), the value computed will be the *concatenation* of the translations of all the elements of the rule with that name. For example, the rule

<A> ::= <B> <C> <D> <C> :{(C *)}.

will assign as the translation of A the concatenation of the translation of the two nodes named C.

In addition to the implicit manipulation of list structure using the parenthetical notation, some LISP functions may be invoked directly to create and decompose list structures. These functions are car cdr cadr cddr caddr append cons append1 gensym

(the list of these functions is stored as the value of the global variable **allowed-lisp-functions** which is defined in file *compiler/sem-compile*). These functions are invoked using standard LISP notation. Such a function call may stand by itself as a translation rule, or may be used as a component of a list created using the parenthetical notation. Each argument to the function can be any element or structure acceptable as a translation rule. For example, if the translation of B is "(ho ho)" and the translation of C is "(quack quack)", the rule

<A> ::= <B> <C> : {((car B) (car C))}.

will compute for A a translation of "(ho quack)".

4. Simplification

The procedure for evaluating translation rules includes a *simplifier* which can perform lambda conversions. The simplifier looks for list structures of the form

((lambda (var) expr) arg)

and performs the lambda conversion, binding variable *var* to *arg*, evaluating expression *expr*, and then replacing the entire structure by the value of the expression. The simplifier also eliminates "extra" parentheses: if the first element of a list is itself a list and is not a lambda expression, the parentheses of the inner list are removed. For example, "((a b) c d)" would be simplified to "(a b c d)".

There are a number of restrictions on the operation of the simplifier. It is not applied at every level of the tree; rather, it is only applied a nodes whose names appear on the list **SIMPLIST** which should be defined as a TYPE list in the grammar. The lambda converter uses a one-pass algorithm, so there are restrictions on the positions of lambda expressions which will be converted. The details of the simplifier are described in the PROTEUS Project Memo *Syntactic Regularization in PROTEUS*.

CHAPTER 7

Conjunctions

The analysis of coordinate conjunction is a difficult problem, and it has a significant impact on any language processing system which aims to treat conjunction with some degree of generality. We describe in this chapter the mechanisms provided in the PROTEUS Parser for coordinate conjunction. These mechanisms are quite simple and quite general, but we do not pretend that they cover the phenomena with the breadth of, say, the Linguistic String Parser.

For the purpose of the presentation here, we divide the task of handling coordinate conjunction into five parts:

(1) creating basic constituent structures for conjunction

(2) constraining these structures to avoid redundant and ungrammatical analyses

(3) expanding (regularizing) conjoined structures

(4) adapting the restrictions to process conjoined structures

(5) adapting the syntactic regularization rules to process conjoined structures

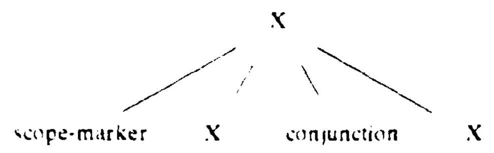
LISP procedures in the PROTEUS parser are provided to create the constituent structure (task 1) and expand "reduced" structures (task 3). The constraints on conjunction (task 2) must be stated as restrictions in the grammar. Adapting the regularization rules (task 5) is achieved largely through the use of the conjunction expansion procedure. This procedure is also of benefit in adapting the restrictions (task 4), although it provides only a very limited solution in this area.

1. The structures

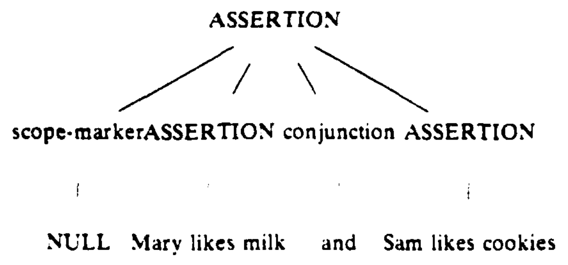
The basic structure provided for conjunction is a symmetric one, with a node of type *X* dominating a two nodes of type *X*, separated by a conjunction and optionally preceded by a scope marker ("both", "either", etc.). In terms of BNF,

<X>::= <scope-marker> <X> <conjunction> <X>

in tree structure



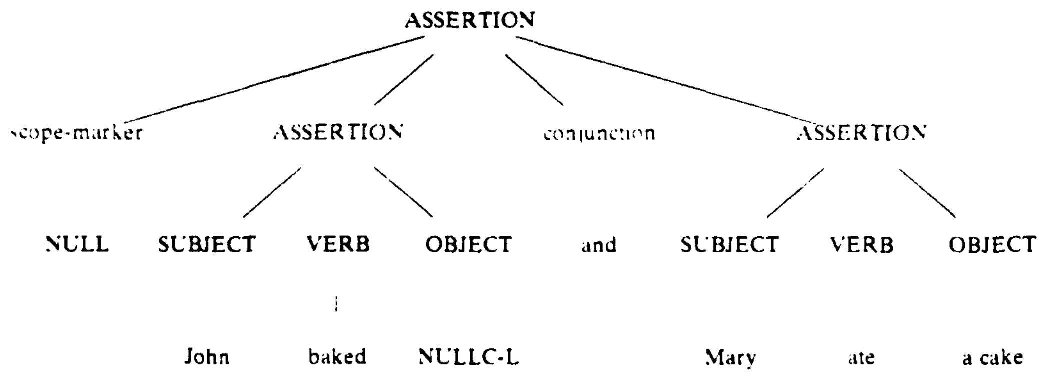
This structure could be used to analyze, for example, "Mary likes milk and Sam likes cookies":



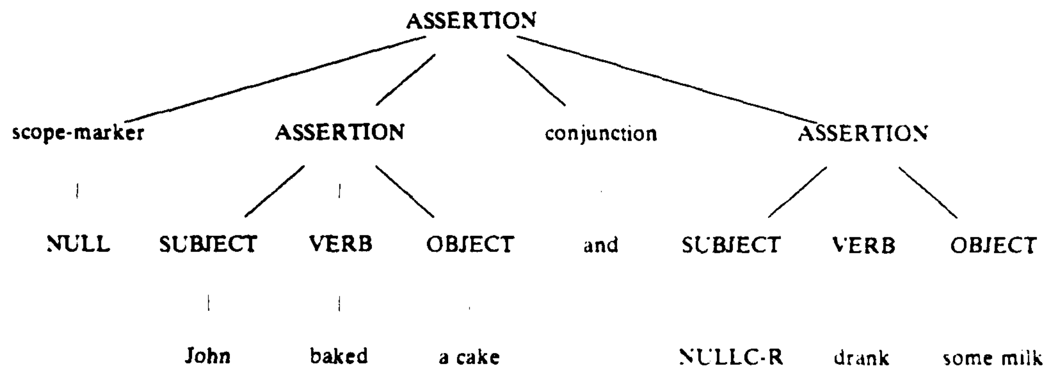
To avoid redundant analyses (for example, that the conjoining in "Sam likes cookies and cake" could be analyzed as N and N, LNR and LNR, NSTG and NSTG, NSTGO and NSTGO, or OBJECT and OBJECT), we restrict conjoinings (the X's above) to members of the STRING and LXR lists.

By itself, this structure is not sufficient to account for sentences such as "John baked and Mary ate a cake." We can describe this sentence as being of the form ASSERTION and ASSERTION, with the OBJECT omitted from the first ASSERTION. Similarly, we could describe the sentence "John baked a cake and drank some milk." as being two conjoined ASSERTIONs, with the SUBJECT omitted from the second ASSERTION. More generally, in a structure of the form X_1 and X_2, we must allow some of the trailing elements of X_1 and some of the initial elements of X_2 to be omitted.

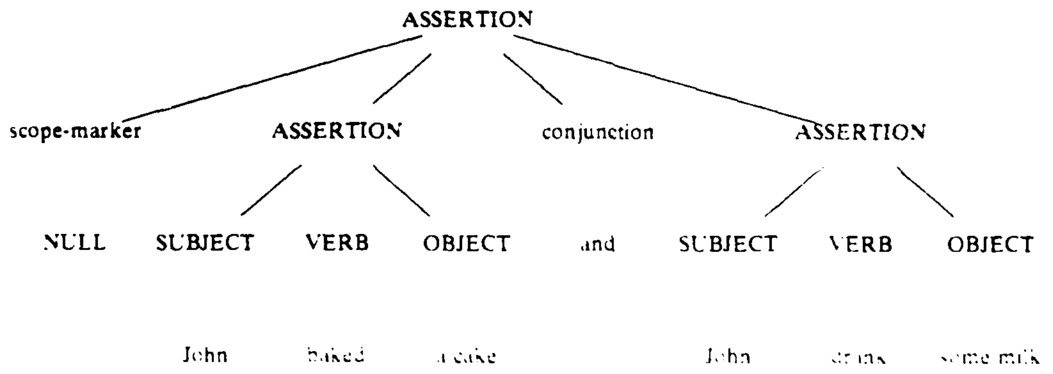
We will represent omission in our parse trees not by actually omitting the element, but rather by giving a node the value NULLC-L (for omission to the left of a conjunction) or NULLC-R (for omission to the right of a conjunction)., Thus "John baked and Mary ate a cake." would be analyzed as



and "John baked a cake and drank some milk." would be analyzed as



For several purposes -- applying restrictions, computing semantics -- we may want to create a more regular structure, without omitted elements. We can create such a structure through a process of *conjunction expansion*, which replaces NULLCs by the value of the corresponding element in the other conjunct. This process would, for example, change the tree just above to



2. Creating conjoined structures

We have chosen to incorporate all the possible conjoined structures into the context-free component, rather than creating them dynamically during parsing (as is done in some ATNs and the Linguistic String Parser). This may be somewhat slower, particularly for sentences without conjunctions, but it keeps the parsing procedure simple.

The parser provides a mechanism for automatically introducing the productions for conjoined structures. The Restriction Language statement

METARULE = metaconj.

adds to the definition of every symbol *X* on the STRING and LXR type lists the option

<SCOPE-WORD> <X> <CONJ-WORD> <X>

These options are added when the grammar is loaded, at the point where the METARULE statement is encountered., The METARULE statement should appear *after* all the non-conjunction-specific restrictions; in this way the housing for these restrictions does not have to be adjusted for conjunction.

The user must provide explicit definitions in the grammar for SCOPE-WORD and CONJ-WORD. In addition, to allow for omissions, NULLC-L and NULLC-R options must be explicitly included (for example, one would add

```
<SUBJECT> ::= ... | <*NULLC-R>
<OBJECT>  ::= ... | <*NULLC-L>
```

to handle the two examples given above).

3. Constraining conjoined structures

The context-free rules by themselves will produce a blizzard of parses, and so must be constrained. In particular, NULLCs must be constrained to occur only in conjoined structures, and only in the correct patterns (e.g., to exclude a single NULL-C in the middle of a string). These constraints must be enforced by restrictions in the grammer.

4. Expanding the conjoined structures

We noted above that, in order to simplify the restrictions and syntactic regularization rules which must apply to conjoined structures, we want first to "expand" these structures, filling in omitted elements with the corresponding element from the other conjunct. The parser provides a built-in routine COPY-CONJ which performs this regularization. This routine is to be executed at the root node of a conjoined structure (a node <X> dominating <SCOPE-WORD> <X> <CONJ-WORD> <X>). It takes no arguments, and so may be invoked

```
CONJ-COPY EXISTS.
```

This routine is unusual in that it is the only operation in Restriction Language which directly modifies the parse tree.

CONJ-COPY relies on two lists: NULLC-LNODES, a list of all symbols in the grammar whose value can be NULLC-L, and NULLC-RNODES, a list of all symbols in the grammar whose value can be NULLC-R. These lists must be given explicitly in the grammar (by statements TYPE NULLC-LNODES = ... and TYPE NULLC-RNODES = ...).

5. Applying restrictions

Restrictions which are executed after CONJ-COPY has been applied will "see" the expanded parse tree, with omitted nodes filled in. However, no mechanism is currently provided for automatically postponing restrictions until CONJ-COPY has applied, or for automatically applying restrictions to each conjunct of a conjoined structure. Thus, accommodating restrictions for conjunction is at present largely the responsibility of the grammar writer.

6. Applying syntactic regularization rules

Associated with each production added by the "metaconj" metarule is a syntactic regularization rule:

```
<X> ::= ... | <SCOPE-WORD><X><CONJ-WORD><X> : (3 2 4)
```

Suppose the CONJ-WORD is "and", and that its regularized form is also "and". This rule would then translate a structure of the form *X1 and X2* into "(and X1' X2'), where X1' and X2' are the translations of X1 and X2.

If the conjoined structure contains an omission, we don't want to compute the regularized form until the omitted elements have been filled in by CONJ-COPY. To do this, the grammar must assign the attribute *CONJOMIT* to nodes which dominate a conjunction omission that has not yet been filled in. If the regularized syntactic structure would normally be computed at a node *N*, but this node has the CONJOMIT attribute, computation of the regularized structure will be postponed (it will get computed as part of some larger structure higher up in the parse tree).

END

10-86

DTIC