PARSING CONJUNCTIONS DETERMINISTICALLY

Donald W. Kosy

The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

ABSTRACT

Conjunctions have always been a source of problems for natural language parsers. This paper shows how these problems may be circumvented using a rule-based, wait-and-see parsing strategy. A parser is presented which analyzes conjunction structures deterministically, and the specific rules it uses are described and illustrated. This parser appears to be faster for conjunctions than other parsers in the literature and some comparative timings are given.

INTRODUCTION

In recent years, there has been an upsurge of interest in techniques for parsing sentences containing coordinate conjunctions (*and*, *or* and *but*) [1,2,3,4,5,8,9]. These techniques are intended to deal with three computational problems inherent in conjunction parsing:

1. Since virtually any pair of constituents of the same syntactic type may be conjoined, a grammar that explicitly enumerates all the possibilities seems needlessly cluttered with a large number of conjunction rules.
2. If a parser uses a top-down analysis strategy (as is common with ATN and logic grammars), it must hypothesize a structure for the second conjunct without knowledge of its actual structure. Since this structure could be any that parallels some constituent that ends at the conjunction, the parser must generate and test all such possibilities in order to find the ones that match. In practice, the combinatorial explosion of possibilities makes this slow.
3. It is possible for a conjunct to have "gaps" (ellipsed elements) which are not allowed in an unconjoined constituent of the same type. These gaps must be filled with elements from the other conjunct for a proper interpretation, as in: *I gave Mary a nickel and Harry a dime.*

The paper by Lesmo and Torasso [9] briefly reviews which techniques apply to which problems before presenting their own approach.

Two papers in the list above [1,3] present deterministic, "wait-and-see" methods for conjunction parsing. In both, however, the discussion centers around the theory and feasibility of parsers that obey the Marcus determinism hypothesis [10] and operate with a limited-length lookahead buffer. This paper examines the other side of the coin, namely, the practical power of the wait-and-see approach compared to strictly top-down or bottom-up methods. A parser is described that analyzes conjunction structures deterministically and produces parse trees similar to those produced by Dahl & McCord's MSG system [4]. It is much faster than either MSG or Fong & Berwick's RPM device [5], and comparative timings are given. We conclude with some descriptive comparisons to other systems and a discussion of the reasons behind the performance observed.

OVERVIEW OF THE PARSER

For the sake of a name, we will call the parser NEXUS since it is the syntactic component of a larger system called NEXUS. This system is being developed to study the problem of learning technical concepts from expository text. The acronym stands for Non-Expert Understanding System.

NEXUS is a direct descendent of READER, a parser written by Ginsparg at Stanford in the late 1970's [6]. Like all wait-and-see parsers, it incorporates a stack to hold constituent structures being built, some variables that record the state of the parse, and a set of transition rules that control the parsing process. The stack structures and state variables in NEXUS are almost the same as in READER, but the rules have been rewritten to make them cleaner, more transparent, and more complete.

There are two categories of rules. *Segmentation rules* are responsible for finding the boundaries of constituents and creating stack structures to store these results. *Recombination rules* are responsible for attaching one structure to another in syntactically valid ways. Segmentation operations are separate from, and always precede, recombination operations. All the rules are encoded in Lisp; there is no separate rule interpreter.

Segmentation rules take as input a word from the input sentence and a *partial-parse* of the sentence up to that word. The rules are organized into procedures such that each procedure implements those rules that apply to one syntactic word class. When a rule's conditions are met, it adds the input word to the partial-parse, in a way specified in the rule, and returns the new partial-parse as output.

A partial-parse has three parts:

1. *The stack*: A stack (not a tree) of the data structures which encode constituents. There are two types of structures in the stack, one type representing clause nuclei (the verb group, noun phrase arguments, and adverbs of a clause), and the other representing prepositional phrases. Each structure consists of a collection of slots to be filled with constituents as the parse proceeds.
2. *The message (MSG)*: A symbol specifying the last action performed on the stack. In general, this symbol will indicate the type of slot the last input word

was inserted in.

3. *The stack-message (MSG1)*: A list of properties of the stack as a whole (e.g. the sentence is imperative).

The various types of slots comprising stack structures are defined in Figure 1. VERB, PREP, ADV, NOTE, and FUNCTION slots are filled during segmentation, while CASES and MEASURE slots are added during recombination. NP slots are filled with noun phrases during segmentation but may subsequently be augmented by post-modifiers during recombination.

CLAUSES	*PREPOSITION STRUCTURES*
VERB: verb phrase	PREP: preposition
ADV: adverbs	ADV: adverbs
NP1,NP2,NP3: noun phrases	NP: noun phrase
NOTE: notes	NOTE: notes
FUNCTION: clause function	
MEASURE: rating	MEASURE: rating
CASES: adjuncts	

DEFINITIONS

Clause function

Hypothesized role of the clause in the sentence, e.g. main, relative clause, infinitive adjunct, etc.

Notes

Segmentation rules can leave notes about a structure that will be used in later processing.

Rating

A numerical measure of the syntactic and semantic acceptability of the structure to be used in choosing between competing possible parses.

Adjuncts

The prepositional phrases and subordinate clauses that turn out to be adjuncts to this clause.

Figure 1: Stack Structures

An English rendering of some segmentation rules for various word classes is given in the Appendix. The tests in a rule depend on the current word, the messages, and various properties of structures in the stack at the time the tests are made. As each word is taken from the input stream, all rules in its syntactic class(es) are tried, in order, using the current partial parse. All rules that succeed are executed. However, if the execution of some rule stipulates a *return*, subsequent rules for that class are ignored.

The actions a rule can take are of five main types. For a given input word W, a rule can:

- continue filling a slot in the top stack structure by inserting W
- begin filling a new slot in the top structure
- push a new structure onto the stack and begin filling one of its slots
- collapse the stack so that a structure below the top becomes the new top
- modify a slot in the top structure based on the information provided by W

In addition, a rule will generally change the MSG variable, and may insert or delete items in the list of stack messages.

The way the rules work is best shown by example. Suppose the input is:

The children wore the socks on their hands.

The segmentation NEXUS performs appears in Fig. 2a. On the left are the words of the sentence and their possible syntactic classes. The contribution each word makes to the development of the parse is shown to the right of the production symbol "=>". We will draw the stack upside down so that successive parsing states are reached as one reads down the page. The contents of a stack structure are indicated by the accumulation of slot values between the dashed-line delimiters ("-----"). Empty slots are not shown.

Input Word	Word Class		MSG1	MSG	Stack

--	--		nil	BEGIN	FUNCTION: *MAIN*
the	A	=>	nil	NOUN	NP1: *the*
children	N	=>	nil	NOUN	NP1': *the children*
wore	V	=>	nil	VERB	VERB: *wore*
the	A	=>	nil	NOUN	NP2: *the*
socks	N,V	=>	nil	NOUN	NP2': *the socks*
					-----
on	P	=>	nil	PREP	PREP: *on*
their	N	=>	nil	NOUN	NP: *their*
hands	N,V	=>	nil	NOUN	NP': *their hands*
					-----

a. Segmentation

{*wear* PN
[SUB *the children*]
[OBJ *the socks*]
[ON *their hands*] }

b. Recombination

Figure 2: Parse of *The children wore the socks on their hands*

Before parsing begins, the three parts of a partial-parse are initialized as shown on the first line. One structure is prestored in the stack (it will come to hold the main clause of the input sentence), the message is BEGIN, and MSG1 is empty. The parsing itself is performed by applying the word class rules for each input word to the partial-parse left after processing the previous word. For example, before the word *wore* is processed, MSG = NOUN, MSG1 is empty, and the stack contains one clause with FUNCTION = *MAIN* and NP1 = *the children*. *Wore* is a verb and so the Verb rules are tried. The third rule is found to apply since there is a clause in the stack meeting the conditions. This clause is the top one so there is no collapse. (Collapse performs recombination and is described below.) The word *wore* is inserted in the VERB slot, MSG is set, and the rule returns the new partial-parse.

It is possible for the segmentation process to yield more than one new partial-parse for a given input word. This can occur in two ways. First, a word may belong to several syntactic classes

and when this is so, NEXUS tries the rules for each class. If rules in more than one class succeed, more than one new partial-parse is produced. As it happens, the two words in the example that are both nouns and verbs do not produce more than one partial-parse because the Verb rules don't apply when they are processed. Second, a word in a given class can often be added to a partial-parse in more than one way. The third and fifth Verb rules, for example, may both be applicable and hence can produce two new partial-parses. In order to keep track of the possibilities, all active partial-parses are kept in a list and NEXUS adds new words to each in parallel. The main segmentation control loop therefore has the following form:

```
For each word w in the input sentence do
    For each word class C that w belongs to do
        For each partial parse P in the list do
            Try the C rules given w and P
            Loop
        Loop
    Store all new partial-parses in the list
    Loop
```

In contrast to segmentation rules, which add structures to a partial-parse stack, recombination rules reduce a stack by joining structures together. These rules specify the types of attachment that are possible, such as the attachment of a post-modifier to a noun phrase or the attachment of an adjunct to a clause. The successful execution of a rule produces a new structure, with the attachment made, and a rating of the semantic acceptability of the attachment. The ratings are used to choose among different attachments if more than one is syntactically possible.

There are three rating values -- perfect, acceptable, and unacceptable -- and these are encoded as numbers so that there can be degrees of acceptability. When one structure is attached to another, its rating is added to the rating of the attachment and the sum becomes the rating of the new (recombined) structure. A structure's rating thus reflects the ratings of all its component constituents. Although NEXUS is designed to call upon an interpreter module to supply the ratings, currently they must be supplied by interaction with a human interpreter. Eventually, we expect to use the procedures developed by Hirst [7]. There is also a 'no-interpreter' switch which can be set to give perfect ratings to clause attachment of right-neighbor prepositional phrases, and noun phrase ("low") attachment of all other post-modifiers.

The order in which attachments are attempted is controlled by the **collapse** procedure. **Collapse** is responsible for assembling an actual parse tree from the structures in a stack. After initializing the root of the tree to be the bottom stack structure, the remaining structures are considered in reverse stack order so that the constituents will be added to the tree in the order they appeared (left to right). For each structure, an attempt is made to attach it to some structure on the right frontier of the tree, starting at the lowest point and proceeding to the highest. (Looking only at the right frontier enforces the no-crossing condition of English grammar.[1]) If a perfect attachment is found, no further possibilities are considered. Otherwise, the highest-rated attachment is selected and **collapse** goes on to attach the next structure. If no attachment is found, the input is ungrammatical with respect to the specifications in the recombination rules.

[1]The no-crossing condition says that one constituent cannot be attached to a non-neighboring constituent without attaching the neighbor first. For instance, if constituents are ordered A, B, and C, then C cannot be attached to A unless B is attached to A first. Furthermore, this implies that if B and C are both attached to A, B is closed to further attachments.

After a stack has been collapsed, a formatting procedure is called to produce the final output. This procedure is primarily responsible for labeling the grammatical roles played by NPs and for computing the tense of VERBs. It is also responsible for inserting dummy nouns in NP slots to mark the position of "wh-gaps" in questions and relative clauses.

Figure 2b shows the tree NEXUS would derive for the example. The code PN indicates past tense, and the role names should be self-explanatory. During collapse, the interpreter would be asked to rate the acceptability of each noun phrase by itself, the acceptability of the clause with the noun phrases in it, and the acceptability of the attachment. The former ratings are necessary to detect mis-segmented constituents, e.g., to downgrade "time flies" as a plausible subject for the sentence *Time flies like an arrow*. By Hirst's procedure, the last rating should be perfect for the attachment of the on-phrase to the clause as an adjunct since, without a discourse context, there is no referent for *the socks on their hands* and the verb *wear* expects a case marked by *on*.

CONJUNCTION PARSING

To process *and* and *or*, we need to add a coordinate conjunction word class (C) and three segmentation rules for it.[2]

1. If MSG = *BEGIN*,
   Push a clause with FUNCTION = *w* onto stack.
   Set MSG = *CONJ* and return.

2. If the topmost nonconjunct clause in the stack has VERB filled,
   Push a clause with FUNCTION = *w* onto stack.
   Set MSG = *CONJ* and return.

3. Otherwise,
   Push a preposition structure with PREP = *w* onto stack.
   Set MSG = *PREP* and return.

The first rule is for sentence-initial conjunctions, the second for potential clausal conjuncts and the third is for cases where the conjunction cannot join clauses. This last case arises when noun phrases are conjoined in the subject of a sentence: *John and Mary wore socks*. Note that the stack structure for a noun phrase conjunct is identical to that for a prepositional phrase.

To handle gaps, we also need to add one rule each to the Noun and Verb procedures. For Verb, the rule is:

4. If MSG = *CONJ*,
   Set NP1 = *!sub*, VERB = *w* in top structure.
   Set MSG = *VERB* and return.

For Noun:

5. If the top structure S is a clause conjunct with NP1 filled but no VERB and there is another clause C in the stack with VERB filled and more than one NG filled,
   Copy VERB filler from C to S's VERB slot
   If C has NP3 filled,
      Transfer S's NP1 to NP2 and set S's NP1 = *!sub*.
   Insert *w* as new NG in S.
   Set MSG = *NOUN* and return.

In both rules, *!sub* is a dummy placeholder for the subject of the

[2]The conjunction *but* is not syntactically interchangeable with *and* and *or* since *but* cannot freely conjoin noun phrases: **John but Mary wore socks*. The rules for *but* have not yet been developed.

clause. Rule 4 is for verbs that appear directly after a conjunction and rule 5 is for transitive or ditransitive conjuncts with gapped verb.

To specify attachments for conjuncts, we need some recombination rules. In general, elements to be conjoined must have very similar syntactic structure. They must be of the same type (noun phrase, clause, prepositional phrase, etc.). If clauses, they must serve the same function (top level assertion, infinitive, relative clause, etc.), and if non-finite clauses, any ellipsed elements (wh-gaps) must be the same. If these conditions are met, an attachment is proposed.

Additionally, in three situations, a recombination rule may also modify the right conjunct:

1. A clause conjunct without a verb can be proposed as a noun phrase conjunct.
2. A clause conjunct without a verb may also be proposed as a gapped verb, as in: *Bob saw Sue in Paris and [Bob saw] Linda in London.*
3. When constituents from the left conjunct are ellipsed, they may have to be taken from the right conjunct, as in the famous sentence: *John drove through and completely demolished a plate glass window.* This transformation is actually implemented in the final formatting procedure since all of the trailing cases in the right conjunct must be moved over to the left conjunct if any such movement is warranted.

Since all these situations are structurally ambiguous, the interpreter is always called to rate the modifications. In situation 2, for instance, it may be that there is no gap: *Bob saw Sue in [Paris and London] in the spring of last year.* In situation 3, the gapped element might come from context, rather than the right conjunct: *Ignoring the stop sign at the intersection, John drove through and completely demolished his reputation as a safe driver.* Hence, only interpretation can determine which choice is most appropriate.

Let us now examine how these rules operate by tracing through a few examples. First, suppose the sentence from the previous section were to continue with the words "and their feet". Rule 2 would respond to the conjunction, and the rest of the segmentation would be:

Input Word	Word Class		MSG1	MSG	Stack
					-----
and	C	=>	nil	CONJ	FUNCTION: *AND*
their	N	=>	nil	NOUN	NP1: *their*
feet	N	=>	nil	NOUN	NP1': *their feet*

Thus, the noun rules would do what they normally do in filling the first NP slot in a clause structure. If the sentence ended here, recombination would conjoin the last two noun phrases, "their hands" and "their feet", as the complement of *on*, producing:

{*wear* PN
[SUB *the children*]
[OBJ *the socks*]
[ON *their hands* (AND *their feet*)] }

If, instead, the sentence did not end but continued with a verb -- "froze", say -- the segmentation would continue by adding this word to the VERB slot in the top structure, which is open. As before, the rules would do what they normally do to fill a slot. Recombination would yield conjoined clauses:

{*wear* PN
[SUB *the children*]
[OBJ *the socks*]
[ON *their hands*]
[AND (V *freeze* PN
[SUB *their feet*]) }

Notice that the second clause is inserted as just another case adjunct of the first clause. There is really no need to construct a coordinate structure (wherein both clauses would be dominated by the conjunction) since it adds nothing to the interpretation. Moreover, as Dahl & McCord point out [4], it is actually better to preserve the subordination structure because it provides essential information for scoping decisions.

Now we move on to gaps. Consider a new right conjunct for our original example sentence in which the subject is ellipsed: *The children wore the socks on their hands and froze their feet.* Rule 4 would detect the gap and the resulting segmentation would be:

Input Word	Word Class		MSG1	MSG	Stack
					-----
and	C	=>	nil	CONJ	FUNCTION: *AND*
froze	V	=>	nil	VERB	NP1: *!sub*
					VERB: *froze*
their	N	=>	nil	NOUN	NP2: *their*
feet	N	=>	nil	NOUN	NP2': *their feet*

Recombination would yield conjoined clauses with shared subject:

{*wear* PN
[SUB *the children*]
[OBJ *the socks*]
[ON *their hands*]
[AND (V *freeze* PN
[SUB *!sub*]
[OBJ *their feet*]) }

The appearance of *!sub* in the second SUB slot tells the interpreter that the subject of the right conjunct is coreferential with the subject of the left conjunct.

Finally, to illustrate rule 5, consider the sentence:

The children wore the socks on their hands and John a lampshade on his head.

When the parser comes to "a", rule 5 applies, the verb *wore* is copied over to the second conjunct, and "a" is inserted into NP2. Thus, the segmentation of the conjunct clause looks like this:

Input Word	Word Class		MSG1	MSG	Stack
					-----
and	C	=>	nil	CONJ	FUNCTION: *AND*
John	N	=>	nil	NOUN	NP1: *John*
a	A	=>	nil		VERB: *wore*
				NOUN	NP2: *a*
lampshade	N	=>	nil	NOUN	NP2': *a lampshade*
					-----
on	P	=>	nil	PREP	PREP: *on*
his	N	=>	nil	NOUN	NP: *his*
head	N,V	=>	nil	NOUN	NP': *his head*

Recombination would produce the conjunction of two complete clauses with no shared material.

RESULTS

Using the rules described above, NEXUS can successfully parse all the conjunction examples given in all the papers, with two exceptions. It cannot parse:

- conjoined adverbs, e.g., *Slowly and stealthily, he crept toward his victim.*
- embedded clausal complement gaps, e.g., *Max wants to try to begin to write a novel and Alex a play.*

The problem with these forms lies not so much in the conjunction rules as in the rules for adverbs and clausal complements in general. These latter rules simply aren't very well developed yet.

It is instructive to compare the NEXUS parser to that of Lesmo & Torasso. Like theirs, NEXUS solves the first problem mentioned in the introduction by using transition rules rather than a more conventional declarative grammar. Also like theirs, NEXUS solves the third problem by means of special rules which detect gaps in conjuncts and which fill those gaps by copying constituents from the other conjunct. Unlike theirs, however, NEXUS delays recombination decisions as long as it can and so does not have to search for possible attachments in some situations where theirs does. For instance, in processing

Henry repeated the story John told Mary and Bob told Ann his opinion.

their parser would first mis-attach [and Bob] to [Mary], then mis-attach [and Bob told Ann] to [John told Mary]. Each time, a search would be made to find a new attachment when the next word of the input was read. NEXUS can parse this sentence successfully without any mis-attachments at all.

It is also instructive to compare NEXUS to the work of Church. His thesis [3] gives a detailed specification of a some fairly elegant rules for conjunction (and several other constructions) along with their linguistic and psycholinguistic justification. While most of the rules are not actually exhibited, their specification suggests that they are similar in many ways to those in NEXUS. However, Church was primarily concerned with the implications of determinism and limited memory, and so his parser, YAP, does not defer decisions as long as NEXUS does. Hence, YAP could not find, or ask for resolution of, the ambiguity in a sentence like: *I know Bob and Bill left*. YAP parses this as [I know Bob] and [Bill left]. NEXUS would find both parses because the third and fifth verb rules both apply when the verb *left* is processed. Note that these two parses are required not because of the conjunction, but because of the verb *know*, which can take either a noun phrase or a clause as its object. Only one parse would be needed for unambiguous variations such as *I know that Bob and Bill left* and *I know Bob and Bill knows me*. In general, the conjunction rules do not introduce any additional nondeterminism into the grammar beyond that which was there already.

With respect to efficiency, the table below gives the execution times in milliseconds for NEXUS's parsing of the sample sentences tabulated in [5]. For comparison, the times from [5] for MSG and RPM are also shown. All three systems were executed on a Dec-20 and the times shown for each are just the time taken to build parse trees: time spent on morphological analysis and post-parse transformations is not included. MSG and RPM are written in Prolog and NEXUS is written in Maclisp (compiled). NEXUS was run with the 'no-interpreter' switch turned on.

Sample Sentences	MSG	RPM	NEXUS
Each man ate an apple and a pear.	662	292	112
John ate an apple and a pear.	613	233	95
A man and a woman saw each train.	319	506	150
Each man and each woman ate an apple.	320	503	129
John saw and the woman heard a man that laughed.	788	834	275
John drove the car through and completely demolished a window.	275	1032	166
The woman who gave a book to John and drove a car through a window laughed.	1007	3375	283
John saw the man that Mary saw and Bill gave a book to laughed.	439	311	205
John saw the man that heard the woman that laughed and saw Bill.	636	323	289
The man that Mary saw and heard gave an apple to each woman.	501	982	237
John saw a and Mary saw the red pear.	726	770	190

In all cases, NEXUS is faster, and in the majority, it is more that twice as fast as either other system. Averaging over all the sentences, NEXUS is about 4 times faster than RPM and 3 times faster than MSG.

CONCLUSIONS

The most innovative feature in NEXUS is its use of only two kinds of stack structures, one for clauses and one for everything else. When a structure is at the top of the stack, it represents a top-down prediction of constituents yet to come, and words from the input simply drop into the slots that are open to that class of word. When a word is encountered that cannot be inserted into the top structure nor into any structure lower in the stack, a new structure is built bottom-up, the new word inserted in it, and the parse goes on. When a word can both be inserted somewhere in the stack and also in a new structure, all possible parses are pursued in parallel. Thus, NEXUS seems to be a unique member of the wait-and-see family since it is not always deterministic and hence need not disambiguate until all information it could get from the sentence is available.

The general efficiency of the parser is due primarily to its separation of segmentation from recombination. This is a divide and conquer strategy which reduces a large search space -- grammatical patterns for words in sentences -- into two smaller ones: (1) the set of grammatical patterns for simple phrases and clause nuclei, and (2) the set of allowable combinations of stack structures. Of course, search is still required to resolve structural ambiguity, but the total number of combinations is much less.

It is not clear whether the parser's speed in the particular cases above comes from divide and conquer or from the differences between Prolog and Maclisp. Nevertheless, as systems are built that require larger, more comprehensive grammars, and that must deal with longer, more complicated sentences, the efficiency of wait-and-see methods like those presented here should become increasingly important.

REFERENCES

[1] Berwick, R.C. (1983), "A Deterministic Parser With Broad Coverage," *Proceedings of IJCAI 8*, Karlsruhe, W. Germany, pp. 710-712.

[2] Boguraev, B.K. (1983), "Recognising Conjunctions Within the ATN Framework," in K. Sparck-Jones and Y. Wilks (eds.), *Automatic Natural Language Parsing*, Ellis Horwood.

[3] Church, K.W. (1980), "On Memory Limitations in Natural Language Processing," LCS TR-245, Laboratory for Computer Science, MIT, Cambridge, MA.

[4] Dahl, V., and McCord, M.C. (1983), "Treating Coordination in Logic Grammars," *American Journal of Computational Linguistics*, V. 9, No. 2, pp. 69-91.

[5] Fong, S, and Berwick, R.C. (1985), "New Approaches to Parsing Conjunctions Using Prolog," *Proceedings of the 23rd ACL Conference*, Chicago, pp. 118-126.

[6] Ginsparg, J. (1978), *Natural Language Processing in an Automatic Programming Framework*, AIM-316, PhD. Thesis, Computer Science Dept., Stanford University, Stanford, CA.

[7] Hirst, G. (in press), *Semantic Interpretation and the Resolution of Ambiguity*, New York: Cambridge University Press.

[8] Huang, X. (1984), "Dealing with Conjunctions in a Machine Translation Environment," *Proceedings of COLING 84*, Stanford, pp. 243-246.

[9] Lesmo, L., and Torasso, P. (1985), "Analysis of Conjunctions in a Rule-Based Parser", *Proceedings of the 23rd ACL Conference*, Chicago, pp. 180-187.

[10] Marcus, M. (1980), *A Theory of Syntactic Recognition for Natural Language*, Cambridge, MA.: The MIT Press.

APPENDIX: SAMPLE SEGMENTATION RULES

WORD CLASS

A: Article

Go begin new np with current word *w*.

M: Modifier

If MSG = *NOUN* and LEGALNP(lastNP + *w*),
Continue lastNP with *w* and return.
Else,
Go begin new np with *w*.

N: Noun

If MSG = *NOUN* & *w* = *that* and lastNP can take a relative clause,
Push a clause with FUNCTION = *THAT*, NP1 = *that* onto stack.
Set MSG = *THAT* and return.

If MSG = *NOUN* or *THAT* & LEGALNP(lastNP + *w*),
Continue lastNP with *w*.
If MSG = *THAT*, set MSG = *NOUN* and return.
If *w* is the only noun in lastNP, return.
If the top clause in the stack has no empty NP, return.

Begin new np:

If MSG = *THAT*,
Replace NP1 with *w*.
Set MSG = *NOUN* and return.

If there a clause C in the stack with NP empty
& C is below a relative clause with VERB filled,
Collapse stack down to C and insert *w* as new NP.
Set MSG = *NOUN*.

If the top structure in the stack has NP empty,
Insert *w* as new NP.
Set MSG = *NOUN* and return.

If MSG = *NOUN* & lastNP can take a relative clause starting with *w*,
Push a clause with FUNCTION = *RC*, NP1 = *w* onto stack.
Set MSG = *NOUN* and return.

If the topmost clause C in the stack has VERB filled,
& C's VERB can take a clausal complement,
Push a clause with FUNCTION = *WHAT*, NP1 = *w* onto stack.
Set MSG = *NOUN* and return.

WORD CLASS

P: Preposition

If *w* = *to* & next word is infinitive verb,
Push a clause with FUNCTION = *INF*, NP1 = *!sub* onto stack.
Set MSG = *INF* and return.
Else,
Push a preposition structure with PREP = *w* onto stack.
Set MSG = *PREP* and return.

V: Verb

If MSG = *BEGIN* & *w* not inflected,
Set NP1 = *YOU**, VERB = *w*, NOTE = *IMP*.
Set MSG = *VERB*, insert *IMP* in MSG1, and return.

If MSG = *VERB* & LEGALVP(VERB + *w*),
Continue VERB with *w* and return.

If there is a clause C in the stack with NP1 filled & VERB empty
& AGREES(*w*,NP1),
If C not top structure in stack, collapse stack down to C.
Set C's VERB = *w* and set MSG = *VERB*.
If C is a subclause, return.

If the top clause C in the stack has NP3 filled,
If C not top structure in stack, collapse stack down to C.
Push a clause with FUNCTION = *THAT*, VERB = *w* onto stack.
Transfer C's NP3 to NP1 of new clause.
Set MSG = *VERB* and return.

If the topmost clause C with VERB filled can take a clause as NP2,
If C not top structure in stack, collapse stack down to C.
Push a clause with FUNCTION = *WHAT*, VERB = *w* onto stack.
If C's NP2 is filled, transfer C's NP2 to NP1 of new clause.
Set MSG = *VERB* and return.

DEFINITIONS

1. The current input word is *w*.

2. The variable lastNP refers to the contents of the last NP slot filled in the top structure.

3. The predicate LEGALVP tests whether its argument is a syntactically well-formed (partial) verb phrase (auxiliaries + verb).

4. The predicate LEGALNP tests whether its argument is a syntactically well-formed noun phrase (article + modifiers + nouns).

5. The predicate AGREES tests whether an NP and a verb agree in number.

6. A structure S "has NP empty" if S is either:
   - a preposition structure with NP empty;
   - a clause with no NP filled;
   - a clause with NP1 filled & VERB filled & either the verb is transitive or it is ditransitive, passive form;
   - a clause with NP1 filled & NP2 filled and verb is ditransitive, not passive form.

7. A relative clause is a clause with FUNCTION = *RC* or *THAT*.

8. A subclause is a relative clause or a clause with FUNCTION = *INF* or *WHAT*.

NOTES

1. Of course, this is just a subset of the rules NEXUS actually uses. Not shown, for example, are rules for questions, adverbs, participles, and many other important constructions.

2. Even in the full parser, there are no rules for determining the internal structure of noun phrases. That task is handled by the interpreter.

3. The noun rules will always insert a new NP constituent into an empty NP slot if such a slot is available. Hence, they will always fill NP3 in a clause with a ditransitive verb, and NP2 in clause which can take a clausal complement, even if these noun phrases turn out to be the initial NPs of relative or complement clauses. Such misattachments are detected by the fourth and fifth verb rules, which respond by generating the proper structures.

4. A clause with FUNCTION = *THAT* represents either a complement or a relative clause. The choice is made when the stack is collapsed.

5. The word *that* as sole NP constituent is either the demonstrative pronoun or a placeholder for a subsequent *WHAT* complement. The choice is made when the stack is collapsed.