A Natural Language System for Music

Brian L. Schmidt

Northwestern Computer Music
School of Music
Northwestern University
Evanston, IL 60201 USA
(312) 491-3895
ihnp4!numusic!brian

1. INTRODUCTION

As computer music systems become more complex and sophisticated, there is often a corresponding increase in the level of expertise required to use them effectively. As a result, a musician typically spends a lot of time "learning the system," instead of making music. What is needed is a user interface which a musician can use with little or no formal training yet sufficiently powerful to handle all of his requests. This paper will describe a natural language interface for musical applications which is both easy to use and sophisticated enough to handle complex applications programs. The system described is designed to be tailored to a number of computer music applications ranging from musical score editing and MIDI recording to digital editing and reverberation control. In its initial application, it serves as a "front end" to *eled*, which serves as a score editor and MIDI sequencer.

2. Why Natural Language?

Natural language has several advantages over conventional input techniques.

- Very easy to learn and use.

  A system which knows the natural language of musicians requires little or no prior knowledge. As a result, the musician need not first translate his request into a less-intuitive, mathematical representation.

- Sufficient power for sophisticated users

  As users gain experience with the system, the system grows in sophistication with them. Thus, there is a natural progression from beginner to experienced user.

- Identical across a wide range of applications

  With a fully integrated natural language system, the view of the system from a user's point of view is the same across all applications. A musician need not learn one syntax for MIDI sequencing, another for score editing, another for digital processing of soundfiles, etc.

- Homogeneous access to tools

  With a natural language system, users have immediate access to all tools at the same level. There is no need to sift through trees or windows as with some graphic interfaces.

- More natural use of intelligent tools

  As more intelligent tools are developed, they can easily be integrated into the environment. The user can then use these tools in the most natural way possible.

- Does not require expensive graphic input devices

  A natural language user interface can run on an inexpensive CRT. This is ideal for low-budget situations like class support.

- Ultimately hands-free operation

  As voice-recognition systems become available, their integration into a natural language system will enable a musician to work with a system without being tied to a standard computer terminal.

3. System Architecture

The natural language system has four main components, a ***dictionary***, an ***internal database***, an ***augmented transition network parser*** (*ATN*) and an ***executer***. Figure 1 shows the overall system architecture.

3.1. The Dictionary

The dictionary contains all words and word types used by the parser. Words in the dictionary include all the verbs (actions) which the system knows about, "music-related" nouns, adjectives, and other miscellaneous words. If a word is not found in the dictionary, it is checked to see if it belongs to one of a number of ***word types***. Word types are special categories of words of which specific examples are either unpredictable or simply too numerous to include every example in the dictionary. Examples of word types are *number (ordinals) , *pitch (any string of characters which parses as a legal pitch name), *file (a unix file), and *soundfile (any NSF soundfile). When the system gets a hard to determine word type (eg. *file, *soundfile, etc.) the word itself gets inserted into the dictionary for the duration of the session. Thus, future uses of the word are handled much more efficiently.

Attached to each word (and type) are features used by the ATN. These include:

- Syntactic category

  The syntactic categories for the word (noun, verb, etc.)

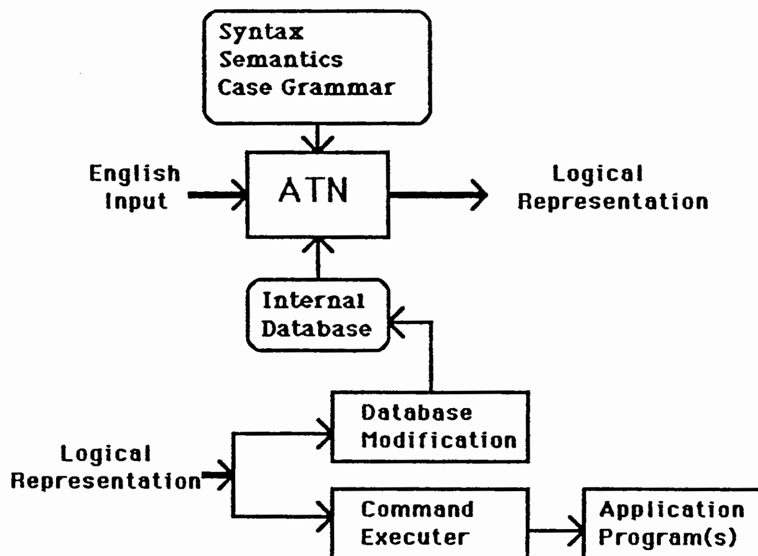

Figure 1. System architecture

- Root

  If the word is not in its root form (infinitive for verbs, singular for nouns), the root is used for syntactic and semantic processing.

- Semantic Class

  Each noun is assigned to one or more *classes* which break the nouns into groups of similar types.

- Miscellaneous Syntactic Information

  Each word can have any number of other features. These include verb tense, number (singular or plural) etc.

- Case Grammar Links

  Each verb is given a link in the *case grammar.* This will be discussed later.

- Semantic Definitions

  Each word has an associated semantic definition used by the semantic functions to build the internal logical representation. This will also be discussed later.

Figure 2 shows some sample dictionary entries.

3.2. The Internal Database

All of the information at the system's disposal is stored in an internal associative database. This includes all instances of nouns the system needs to know about (for example, instruments, channels, soundfiles, etc.), and any other information necessary (i.e. time signature markings, phrase markers, etc.). All information is stored in a predicate calculus representation. For example,

(Instance bass-3 bass)

Represents the fact that there is an instance of a bass, which is known to the system as bass-3. Likewise,

(Instance flute-16 flute)
(Quality flute-16 good)

represents some flute, known to the system as flute-16, which is good in quality. To access the internal information, we use a unification-based retrieval function with pattern-matching variables. For example, the request

(retrieve-val ?x (Instance ?x bass))

will return a list of all the instances of "bass" in the database. By using a conjunctive request, a more selective database retrieval may be done. For instance, if we need to find "the good flute parts," we make a request of the database as follows:

```
(retrieve-val ?x
    (and (Instance ?x flute)
        (Quality ?x good)))
```

This will return a list of only those flutes in the internal database which have a quality good.

bass

category	(noun adjective)
root	bass
class	(instrument range)
numb	singular

second

category	(noun cardinal)
root	second
class	(musical-interval time-interval)
numb	singular

transpose

category	(verb)
root	transpose
tense	tenseless

Figure 2. Sample dictionary entries

The database retrieval system is augmented with simple deductive capability. Thus, an item not actually in the database will be retrieved if that item can be inferred from items (data plus rules) which are in the database. The deduction is done by *backward chaining*, in which one fact can imply the existence of another directly. This is most useful when a class of objects is a subclass of another, and we would like to inherit the attributes of the main class without explicitly entering the facts into the database. For example, to process a noun phrase like "the brass parts," we would expect the database query to be something like this:

(retrieve-val ?x (Instance ?x brass))

and expect it to return all instances of "brass" in the database. Our database, however, has no direct instances of "brass"; instances of instruments are stored as specific instrument names, like

(Instance tuba-1 tuba)
(Instance trumpet-18 trumpet)

etc. As a result, the above database query would not succeed. To fix this, we add the backward chaining rules like "if ?x is an instance of a trumpet, then ?x is an instance of brass instruments," and so on. We can do this by adding the following rules with the special *antecedent/consequence* predicate, "<-" to the database.

(<- (Instance ?x brass) (Instance ?x trumpet))
(<- (Instance ?x brass) (Instance ?x tuba))

With these rules, any query of "brass" of the database will return all instances of trumpets and tubas.

3.3. The Parser

The user's input is parsed by an *augmented transition network parser*. The ATN parser is a state network parser which consists sets of *sub-networks*. These sub-networks consist of *state nodes* and *syntactic match arcs* augmented with context-dependent rules and actions, and semantic processing. Figure 3 shows a simple noun phrase sub-network of the ATN. As each word is processed its syntactic category is matched against the match arcs from the current state. A match is made if any of the following hold true for the current state:

- the syntactic category of the word matches that of the arc
- the arc is labeled "jump" or "send"
- the arc is labeled with another sub-network name which legally parses the next few words

In addition to the syntactic match, there may be any number of other conditions which must be met for the arc to be taken. These are typically context dependent additions to the grammar. If there is one match, that arc is taken and processing continues; if more than one arc match, one is chosen at random and taken while the other matching arcs are saved along with the current state on the *back-tracking list*; if no arcs match and the back-track list is not empty, the first element is removed and processing continues from that state; if none of the arcs match and the back-tracking list is empty, then the parse fails. When an arc is taken, there are typically a number of actions which go with the arc. These actions typically alter the input stream or modify *registers* which hold contextual information (a *transformation*), or perform semantic processing. As the sentence is parsed, a *parse tree* is created. The parse tree represents the syntactic structure of the input sentence, and is used by the ATN to remember grammatical context. Figure 4 shows a typical parse tree.

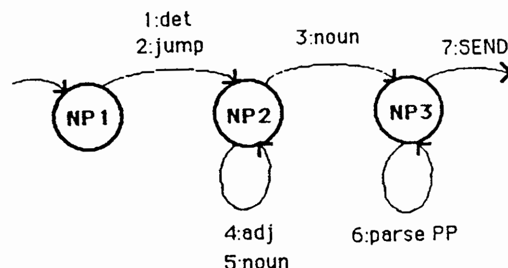

Figure 3. Noun-phrase sub-network of the ATN

3.3.1. Case Grammar Processing

While syntactic processing breaks up the sentence into structural units and phrases, it in itself is an incomplete picture of the meaning of the sentence. For example, consider the following two simple sentences:

Play the bass.

Play the coda.

In each case, the noun phrase is the direct object of the verb. However, the type of action requested for each sentence is very different; the former is requesting a single instrument be played, while the latter is requesting a specific section. The applications program most likely has a different method for accomplishing each. To distinguish between these two cases, *case grammar* routines are used. Case grammar routines supply the information necessary to determine exactly how the syntactic units relate to the action being performed i.e. the main verb. For each verb, a case grammar is defined. The case grammar tell specifically how the various noun phrases and prepositional phrases relate to it depending on the syntactic function, the preposition (if any) and the semantic class of the noun. For example, the case grammar for the verb "play" might be defined as follows as in figure 5.

Thus, "play the bass" would translate into something like

(Instrument (Action play) (the bass)),

while "play the coda" would become

(Section (Action play) (the coda)).

This makes it possible for the executer to create the appropriate command depending on the meaning of the sentence. Finally, the case grammar makes a check of semantic plausibility of the sentence. Consider the following syntactically valid, yet anomalous sentence:

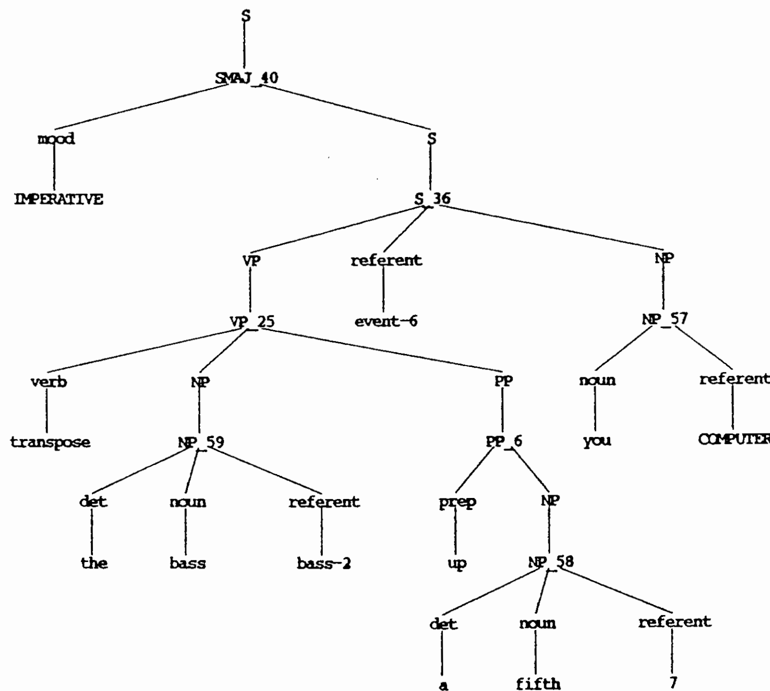

Figure 4. Parse tree for "transpose the bass up a fifth"

transpose the bass up a coda.

The case grammar routines would mark this as anomalous by noting that "transpose," when modified by a prepositional phrase beginning with "up," requires the noun of the prepositional phrase be of semantic class musical interval. Thus, "up a coda" can not modify "transpose," since "coda" is of semantic class "musical section."

	Play	
Preposition	*Semantic Class*	*Case*
from	musical-section	StartTime
from	beat-reference	StartTime
to	musical-section	EndTime
direct-object	instrument	Instrument
direct-object	musical	section
direct-object	beat-reference	StartTime

Figure 5. Case grammar for the verb, "play"

3.3.2. Noun Phrase Semantics

Consider the sentence, "play the first trumpet part." Using the case grammar routines, we have determined that the action, play, is to be carried out on the musical instrument described by the noun phrase, "the first trumpet." However, we do not know how the specific noun phrase translates into an internal representation suitable for use by the applications program. The semantic routines do this by constructing the *sense* of the noun phrase from the words, and by determining the *referent* of the noun phrase from the sense. The referent will eventually be bound to internal representation which the applications program will use. Let's suppose the internal database holds the following information:

(Instance trumpet-16 trumpet)
(Number trumpet-16 1)
(Instance trumpet-23 trumpet)
(Number trumpet-23 2)

representing two trumpet parts, first and second, which have representations in the applications program, trumpet-16 and trumpet-23 respectively. When the parser is parsing the word "first," it executes the semantic definition for "first." This adds the database query

(Number ?x 1)

to the sense of the noun phrase. When the semantics of "trumpet" is processed, the semantics of "trumpet" is executed, adding

(Instance ?x trumpet)

to the sense of the noun phrase. At this point, the sense of the noun phrase is

(and (Instance ?x trumpet)
(Number ?x 1))

Finally, the semantics of "the," which had been put on hold when it was parsed, are executed. The semantics of "the" create the appropriate database query procedure out of the sense of the noun phrase. So, the sense of "the second trumpet" becomes

```
(setf (the referent of %NP)
    (retrieve-val ?x (and (Instance ?x trumpet)
                          (Number ?x 1))))
```

where %NP is the current noun phrase. Finally, the sense of %NP is executed, setting the referent to the value returned by retrieve-val, in this case, trumpet-23.

Just as with the dictionary, there are a number of word classes whose members are too numerous to list individually in the internal database. For example, "measure 5" or "beat 44.5" are both typical noun phrases. However, it would be impossible to represent every conceivable score access point in the internal database. To solve this problem, we augment the semantic definition of words whose referents are not likely to be found in the internal database. In addition to the semantic definition, we include the name of the procedure for determining the referent. We then change the semantic definition of determiners (the, a, etc.) so that if the noun phrase referent is not to be found in the internal database, it uses the procedure specified by the semantics of the noun.

3.3.3. Sentence Semantics

The sentence semantic functions create the final internal representation of the input sentence. Each syntactic unit (subject, direct-object, verb, etc.) has an associated semantic definition. After a sentence is syntactically parsed, sentence semantic routines construct the sense of the sentence by combining the senses of each syntactic unit. Consider the sentence, "play the first phrase of the bass." The semantics of "verb" say to create a new event, say event-33 whose action is play. The sense of the sentence then becomes

```
(Action event-33 play)
```

The semantics of "direct-object" say to use the case grammar routines and add the results to the sense of the sentence. Assuming "the first phrase" is beats 24 to 48, the sense of the sentence is now

```
(and (Action event-33 play)
    (StartTime event-33 24.0)
    (EndTime event-33 48.0))
```

Likewise, the semantics of "prepositional phrase" cause the case grammar representation of "of the bass" to be added to the sense of the sentence. Assuming that **bass-44** is the referent of "the bass," the sense now becomes

```
(and (Action event-33 play)
    (StartTime event-33 24.0)
    (EndTime event-33 48.0)
    (Instrument event-33 bass-44))
```

Finally the semantic routines check the mood of the sentence as determined by the parser (imperative or declarative). The semantic definitions of each mood describe how the sense of the sentence is to be executed. In the case of declarative sentences, like "the rev-7 is on effects send 4," the sense is asserted into the internal database. For imperative sentences, the sense serves as the argument to the executer procedure, which sends the appropriate commands to the applications program. Figure 6 shows some sample sentences and their corresponding internal representations.

play the bass, flute and drums.

```
(Evaluator '(and
    (Action event-1 Play)
    (Agent event-1 COMPUTER)
    (Instrument event-1 (bass-16 flute-24 drums-22))))
```

transpose the first phrase of the koto up a perfect fifth.

```
(Evaluator '(and
    (Action event-2 Transpose)
    (Agent event-2 COMPUTER)
    (Instrument event-2 koto-44)
    (MInterval event-2 +7)
    (StartTime event-2 0.0)
    (EndTime event-2 36.0)))
```

the plate is on send 2.

```
(assert '(and
    (Effect effect-1 plate)
    (EffectAssignment effect-1 Send-2)))
```

Figure 6. Sentences and their corresponding internal representations

3.3.4. Resolving Syntactic Ambiguity

Consider the sentence, "show me the first trumpet and flute parts in measure 16." From a syntactic point of view, there are several interpretations of this sentence. The prepositional phrase, "in measure 16" can be interpreted as modifying either the verb, "show," the noun phrase, "the first trumpet and flute parts," or just "flute parts." Also, the ordinal, "first" can modify either "trumpet and flute parts," or just "trumpet (part)." To decide which of many possible syntactic parses represents the user's intentions, as the sentence parsed, it is given a *likelihood rating*. The likelihood rating is an integer representing the system's ranking of plausibility of user intention. A higher score represents a more plausible parse; a negative value means the system could not make sense of the sentence, even though it passed the syntactic parser. A parse's score can be modified in a number of different ways. For example, if any noun phrase has no referent, i.e. there is no item in the database corresponding the the noun phrase, the likelihood rating is decreased by a very large amount, and the parse is marked describing the error. Also, parses which have prepositional phrases which match the case grammar of the verb are given a higher likelihood rating than parses which have no matches. Thus, the parse in which "in the coda" modifies the verb, "show" will be considered more likely than any other.

When all the parses for a sentence have been finished, the one with the highest likelihood rating is taken and passed to the executer. In the case of a tie, the user is informed of the ambiguity and asked to clarify his intentions. If none of the parses has a non-negative rating, the error message(s) from the "best" parse are given to the user; the parse is not given to the executer.

3.4. The Executer

The executer carries out all of the user's command requests. It takes the internal representation created by the parser and creates appropriate commands for the applications program. The executer has two main sections, a *bulletin board*, and a set of *application functions*. The bulletin board is used to maintain a working context of user requests; the application functions read the bulletin board and create the appropriate command for the applications program. When the executer gets an internal representation from the parser, it posts the representation on the bulletin board according the the *posting function* associated the each verb. It only changes those items which are specifically set by the user's input sentence. In this way, certain parts of the bulletin board change while others may remain unchanged. This allows the user to maintain a continuing dialog with the system allowing partial specification of commands where context makes them clear. For example, after the command "play the bass from measure 24" the bulletin board will look something like this:

Verb	**play**
Command	**Play**
Instrument	**bass-35**
StartTime	**96.0**

If the next command is "add the drums and guitar," the posting function for "add" is used by the executer to post the results of the parse. For add, the posting function says to leave the "Command" slot of the bulletin board unchanged, and add the other cases where appropriate. This will result in the following bulletin board configuration:

Verb	**add**
Command	**Play**
Instrument	**(bass-35 drums-16 guitar-4)**
StartTime	**96.0**

Input	System Output	Comments
show me the bass part.	t [M "bass-16"]	*All events of group "bass-16" are typed*
record a new flute part.	s rchan 7 a 0 0 mv 7 flute_1 . o {flute-24}	*First, set the record channel to the next available midi channel. Then use the add command to add the proper voice change into the score. Finally, using the overdub command, a flute part can be entered on the midi keyboard. The new flute, flute-24, is also entered into the internal database.*
give me more bass in the headphones.	I don't know what "the headphones" is.	*No referent was found for the noun-phrase, "the headphones."*
play the bass.	0,$p [M "bass-16"]	*The bass part plays.*
add the drums.	0,$p [M "bass-16" or M "drums-22"]	*The bass and drums play.*
the guitar.	0,$p [M "bass-16" or M "drums-22" or M "guitar-2"]	*The implied verb, "add," is clear from context.*
again	0,$p [M "bass-16" or M "drums-22" or M "guitar-2"]	*A user can repeat the previous request with "again."*
transpose measures 24 to 30 of the woodwinds up an octave.	96,120e (p "pitch" += 12) [M "flute-24" or M "oboe-17"]	*The flute and oboe are inferred to be "the woodwinds."*

Figure 7. A dialog with the natural language system using eled

As a result, when the application function reads the bulletin board, it creates the command to play the bass, drums and guitar from measure 24.

Each verb the system understands has an associated application function. This function is called by the executer for whatever is in the "Command" slot on the bulletin board. It contains all the information necessary to construct the proper command to be sent to the applications program from the information on the bulletin board. In the case of score editing, these programs create ascii command lines and send them to *eled*, which executes them. Figure 7 a section of a dialog with the natural language system, and the *eled* commands each sentence creates.

4. Support Tools

The system includes a set of support tools to help the system administrator in debugging new applications. The dictionary editor allows the administrator to add words to the system and modify their syntactic and semantic properties. A grammar editor is included to help in the lengthy process of specifying the syntactic grammar which the system understands. In addition to the editor, a number of graphic support routines are included for displaying the grammar graphically and for graphing parse trees of individual sentences.

5. Plans for Future Work

At the writing of this paper, the natural language processor has taken English input and created *eled* commands similar to those presented. Future planned applications include incorporation into a digital mixing/editing environment for NCM and a timbre design system using KSM.

The major problem with the system as it currently exists is its limited knowledge of the musical domain. As a result, user requests must be explicitly stated, using specific instructions and commands. For example, a command like "take it from the c chord in the second verse" poses several problems for the system. Firstly, "the second verse" would have to have been defined explicitly by the musician using the system in terms of either beats or measures; if measures are used, he must specify the time signature(s). Secondly, since the system has limited access to the score on a micro level and since it has no knowledge of harmony, the system would not be able to determine a referent for "the c chord," and all parses would have a negative likelihood rating. What is needed is access to a large base of musical knowledge which can examine the score and return its findings to the system. Current plans are to incorporate the rule-based Knowledge System for Music (KSM) into the natural language system. KSM will act as an augmentation to the internal database deduction system. If no referent is found for a noun phrase, the system will check to see if the referent can be found using the rule-sets of KSM. If so, control will be handed to the knowledge system to determine the proper referent. KSM, along with sophisticated sets of rules, will add musical "intelligence" to the system. These rule-sets will include easy rules-sets, such as for chord and note finding, and more complex ones, like phrase and meter determiners. Figure 8 how KSM and the natural language system will eventually communicate.

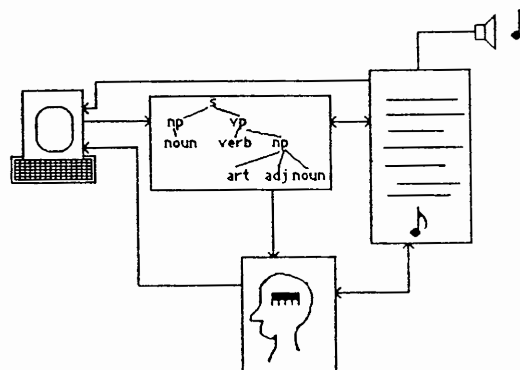

Figure 8. Adding rule-based knowledge to the natural language system

Acknowledgements

I would like to express appreciation for the support from the Systems Development Foundation for the support which made this work possible. I would also like to thank the Gary Kendall and the staff at NCM for support and ideas. Also, special thanks to my thesis advisor, Richard Ashley, and to Derek Ludwig for his programming and graphics help.

References

Ashley, R.D. (1985) KSM: An essay in knowledge representation in music. In B. Truax, ed., *Proceedings of International Computer Music Conference 1985* pp. 383-390. San Francisco: Computer Music Association.

Charniak, E. and McDermott, D. (1985) *Introduction to Artificial Intelligence* Addison-Wesley.

Chomsky, N. (1957) *Syntactic Structures.* Mouton, The Hague.

Decker, S., and Kendall, G.L. (1985) *A unified approach to the editing of time-ordered events* In B. Truax, ed., Proceedings of International Computer Music Conference IX pp. 69-77. San Francisco: Computer Music Association.

Fillmore, Charles J. (1968) "The case for case," in *Universals in Linguistic Theory,* ed. E Bach and R. T. Harms, Holt, Reinhart and Winston, New York, 1-90.

Harris, M. (1985) *Introduction to Natural Language Processing,* Reston Publishing Co., Reston.

Wilensky, R. (1984) *Lispcraft* pp. 287-318. W. W. Norton & Co.

Winograd, Terry (1983) *Language as a Cognitive Process: Vol. 1 Syntax* Addison-Wesley.

Woods, W. (1972) "An experimental parsing system for network grammars," in *Natural Language Processing,* ed. R. Rustin, pp. 111-154. Algorithmics Press, New York.