THE INDUCTION OF THE SYNTAX OF NATURAL LANGUAGE BY COMPUTER

by

Paul James Orgren

Computer Sciences Technical Report #378

December 1979

THE INDUCTION OF THE SYNTAX OF NATURAL LANGUAGE BY COMPUTER

BY

PAUL JAMES ORGREN

A thesis submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1979

THE INDUCTION OF THE SYNTAX OF
NATURAL LANGUAGE BY COMPUTER

Paul James Orgren

Under the supervision of Professor Leonard Uhr

ABSTRACT

This thesis discusses a computer program, Synner, that attempts to learn an augmented transition network (ATN) grammar for English. The process being modelled is that of a human learner who already has some conceptual knowledge and a vocabulary relating to that knowledge. This learner observes a real-world event and hears a sentence describing that event, and then adjusts his rules of grammar, if necessary, to account for the new sentence.

Synner's conceptual knowledge and vocabulary are predefined in terms of conceptual dependency (CD) theory, a representation based on a small number of primitives. To avoid the problems of perception, the "real-world" event is hand-coded into a CD structure, while the sentence is given as a string of words on an interactive computer terminal.

The learning procedure begins by comparing the sentence to the event using its built-in knowledge of the vocabulary. This matching process determines the semantic role of each word in the sentence. A parse of the sentence is then attempted using the current ATN grammar. There is a set of rules specifying how the sentence is allowed to differ from what is allowed by the grammar. The search for differences proceeds from the smallest ones to larger ones, in order to insure that the least possible difference that can account for the sentence is found. The differences discovered, if any, are used in altering the ATN so that the new sentence is legal.

The conciseness of the ATN is dependent on the training sequence used. A sequence of sentences that builds slowly on previous sentences results in a more compact grammar than does a sequence that jumps from one type sentence to another. The results are encouraging for research along these lines, with as little as possible presupposed about the structure of the grammar to be learned.

ACKNOWLEDGMENTS

I wish to thank Professor Leonard Uhr for his guidance in all stages of this research. I would also like to thank the other members of the committee, Professors Gregg Oden, Larry Travis, William Havens and Charles Davidson, for their helpful comments. Thanks are also due Greg Scragg for his advice during the early stages of this work.

I wish to thank my parents, James and Sally Orgren, for instilling in me the desire for education. Most of all, I would like to thank M. Cecilia Ferreira-Orgren, without whom this dissertation would never have been done; and to Lisa Orgren, for bringing a lot of joy to our lives. To Elsy Ferreira, many thanks for the help at home so that I could spend more time on this research during the final stages.

Thanks to Hideko Takahashi for drawing the figures, and to Jess Anderson for the use of his terminal.

This research was supported in part by a grant from the National Science Foundation to Professor Leonard Uhr.

Table of Contents

List of Figures

CHAPTER 1
INTRODUCTION

This thesis investigates the question of how a grammar for a natural language can be inferred by a computer, given a semantic structure, a dictionary of words defined in terms of that structure, and a series of grammatical sentences together with an event for each sentence upon which the meaning of the sentence is based.

History

Several researchers have investigated the problems of natural language learning by computer. The work of some of these will be reported below.

Uhr (1964) was among the first to deal with the question of natural language learning (as opposed to natural language processing) by computer. He discussed two programs and a projected third program which learn to translate from one language to another. These programs do not have a semantic memory. The learning process operates only on strings: the teacher types in a sentence in one language, the program replies with its best attempt at translation, and then, as feedback, the teacher provides the desired response. Each program looks for matches between the feedback string and its own output string, and increases the weights of rules that led to matches and decreases the weights of rules that caused mistakes. As the weights are adjusted over the course of several training sentences, the program can handle more sentences correctly.

The first of Uhr's programs operates on strings of words separated by blanks. The second program is given strings with no word delimiters; one of the things it has to learn is how to find the words. The third program was to handle words in context, making decisions based on groups of words rather than one word at a time.

Quillian (1969) developed a program called the Teachable Language Comprehender (TLC) that learns facts which are used in "comprehending" English text. TLC has a semantic network representing factual assertions about the world. Text is presented to the program, which "comprehends" that text by relating each assertion (whether explicit or implicit) it contains to the semantic memory. TLC then uses its comprehension to adapt the semantic network, incorporating the facts represented by the text. These new facts can be used to aid the comprehension of future sentences.

Klein et al. (1968) described the Autoling system, which builds context free phrase structure grammars. It operates only on strings of words -- no semantic referent is used. Autoling receives grammatical strings from its informant, compares the current string with the current grammar, and hypothesizes a change in the grammar to account for the new sentence. It generates a sentence based on this change, and asks the informant whether it is a correct sentence. If not, the change is not made and a different one is tried. It thus uses negative feedback to guide the learning process.

Autoling is meant as a model of a linguist constructing a grammar for a language rather than as a model of a child learning a language. Linguists use negative information for faster and more accurate construction of the grammar. However, psychological studies (Brown, 1973) have shown that children get little negative feedback concerning syntactic errors and make little use of any they do get.

Jordan's (1972) METQA is similar to Uhr's (1964) program in that the language learning is based on pattern recognition. METQA is not given any built-in semantic or linguistic information. It receives input in the form of unstructured, unsegmented strings and gets feedback interactively from a human trainer. This learning mechanism is used to learn several types of language behavior within the same program: translation, information organization, and question answering.

The language learning that Siklossy's (1972) ZBIE program does is based on the "Russian through Pictures" book (Richards, 1961), where a human learner is given a series of pictures along with sentences describing those pictures. In place of the pictures, ZBIE is given a functional language (FL) internal representation of those pictures. By matching the FL structures with the corresponding natural language sentences, ZBIE constructs a set of rules for translating from FL into Russian. The induced rules are pattern matching rules similar to those used by Uhr (1964).

Salveter (1979) has implemented a program (Moran) which learns Conceptual Memory Structures (reminiscent of Shank's Conceptual Dependency structures, discussed in chapter 2) from perceptual input. Moran's input is a series of two snapshots that depict an action occurring and a sentence that describes that action. The snapshots and sentences are provided by a human trainer. The output is a set of conceptual graphs that represent the various meanings that were discovered for each root verb, and the relationship among these meanings.

For each snapshot sequence, Moran first creates a representation for what happened by linking states in the first snapshot with states in the second snapshot. It then locates the existing representation that it considers to be closest semantically to the current input. Finally, it modifies the graph structure so that the current input is represented and similarities to and differences from previous inputs are reflected.

Anderson (1977) investigated the induction of augmented transition networks (ATNs) for natural language (see Chapter 2 for a discussion of ATNs). His Language Acquisition System (LAS) uses a version of HAM (Anderson and Bower, 1973) for its knowledge structure. LAS is presented with a series of sentence-event pairs, where the event represents the meaning of the sentence and is specified as a HAM structure.

The HAM structure is a propositional network that contains English words to represent the objects and actions involved in a proposition. In order to construct an ATN, the HAM event structure is manipulated to match the sentence. This manipulation involves bending and twisting the graph until it has the meaning-bearing words of the sentence in the same order as they appear in the sentence. If this reordering can be accomplished without any branches crossing, the graph deformation condition is said to be met, and learning can take place. When the graph deformation condition is met, the BRACKET routine is activated. BRACKET is a routine which uses the results of the graph deformation to form the surface structure of the sentence; it also leaves room for pre-positional and post-positional noun modifiers, whether or not they exist in the sentence. For example, the bracketed structure for "The girl hit the boy who liked the cake" would be:

((The()girl())hit(the()boy(who liked(the()cake()))))

BRACKET has determined the entire structure for the sentence, including the structure of adjectives (which do not even appear in the sentence) for "girl", "boy" and "cake". This bracketed sentence is then used in building the ATN; the levels of bracketing indicating the level of the subnets to be formed. Because of the way BRACKET works, LAS is apparently being given most of the rules of English syntax; the training sequence is used merely to convert those rules into the form of an ATN.

Because of the restrictions on graph deformation, LAS cannot handle situations where semantically related components are separated in the sentence. These situations can occur in inflected languages like Latin, and occur in English in "respectively" constructions, e.g.

"John and Bill ate ice cream and cake, respectively."

Goals of the Present Research

There is no algorithm that can be guaranteed to learn any arbitrary grammar from the class of finite state grammars in a finite amount of time (Gold, 1967; Anderson, 1976). However, humans do learn natural language, which has at least the complexity of a finite state grammar. Natural languages, therefore, must not be arbitrary selections from a class of languages, but must be restricted in some way such that human learners can use a set of heuristics in discovering the grammar. These heuristics would not work for all grammars, but would enable learning of those that met the constraints. Natural languages must meet these constraints (whatever they are) in order to be learnable.

Studies by Moeser and Bregman (1972, 1973) have shown that for humans, semantics makes learning a grammar easier. In one of their experiments, two groups of subjects were given 3000 training sentences in a natural-like artificial language; one group had pictures for a semantic referent, the other had no semantic aids. In a test given after the training sequence was completed, the group that had seen only strings was very close to chance in judging the grammaticality of strings, while the group with the semantic referents during training (though not given any referents for the test strings) was almost perfect.

The research described in this thesis assumes that semantics can guide the search for grammars of human languages. It investigates how a computer program might induce an augmented transition network for English, given sentence-event pairs, where the event is a conceptual dependency structure upon which the sentence is based. For example, the sentence

"The big girl hit the boy."

would be presented along with an associated event which shows a girl hitting a boy. This is similar to what Anderson (1977) did; the main difference is in the semantic representation. The HAM structure Anderson used is very close to English. Also, his program used a bracketing procedure which built in much knowledge of what a grammatical sentence should be.

A computer program, Synner, has been implemented to investigate the mechanisms a program might use to learn an ATN grammar from a series of conceptual dependency-sentence pairs. Its behavior indicates that a few well chosen rules on using the semantic information can result in a concise grammar that handles the sentences seen, given a favorable training sequence.

Overview

Before any learning is done, Synner has a built-in dictionary which defines words in terms of a Conceptual Dependency(CD)-like semantic structure (see chapter 2). Learning takes place when Synner "observes" an event and "hears" a sentence describing that event. To avoid the problems of perception, the event is pre-coded in the same CD-like structure that is used for the permanent dictionary. To avoid the problems of speech recognition, the sentence is given as a string of written words on a computer terminal.

Synner uses its dictionary to compare the words in the sentence with the event that is given, and determines the semantic role of each of the words in the sentence. It uses this match information in constructing and modifying an ATN grammar that describes the structure of the sentences that have been seen. When each sentence has been used, it is discarded and Synner proceeds to the next sentence-event pair.

Organization of the Dissertation

Chapter 2 discusses the Conceptual Dependency theory of Schank and the Augmented Transition Network formalism of Woods. Both of these representations are heavily used by Synner. Chapter 3 describes the learning mechanisms used by Synner. Chapter 4 illustrates results obtained from one of Synner's learning sessions. Chapter 5 discusses the implications of this research and possible future directions.

CHAPTER 2
REPRESENTATIONS

Conceptual Dependency

Schank (1972) describes a theory of knowledge representation called Conceptual Dependency (CD). It is intended as a language independent way of representing the meaning underlying natural language. CD is a verb based structure that is supposed to represent most actions describable by language.

In CD, all verbs are described in terms of a small number (10 - 15) of primitive ACTs. The primitives are supposed to represent concepts that humans learn early in life and upon which more complex conceptualizations are built. The exact set of primitives used in CD changed from one paper to another as they were debated among Schank's students, but the basic idea remained the same. One possible set of primitives is the following (these are not to be construed as having the full range of meaning of the English words that they resemble): PROPEL, MOVE, INGEST, EXPEL, GRASP, PTRANS, ATRANS, CONC, MTRANS, SPEAK, SENSE. Briefly, PROPEL is one entity imparting motion to another; MOVE means moving oneself or one's body part(s); INGEST is to take something into the body; EXPEL is to expel from the body; GRASP is what a hand can do; PTRANS is a change in location; ATRANS is a change in ownership; CONC is to conceptualize or create ideas; MTRANS is to move information from one part of the mind to another; SPEAK is obvious; and, finally, SENSE is used with a modality to represent seeing, hearing, smelling, etc. Since Synner is concerned solely with physical events, only the first six primitives are used by Synner.

Also involved in CD theory are "picture producers" (PPs -- essentially, physical objects) and "picture aiders" (PAs) that describe PPs. PAs can be states or relations. States are things like FEAR, HUNGER, COLOR, etc. that have a scalar value. Relations connect two or more PPs; for example, one PP can be PART-OF another or in PHYSICAL-CONTACT with another.

The primitive ACTs, the PPs and the PAs are joined together using connectors drawn from a small set (10 or so members). Some of these connectors and the rules for using them are shown in Figure 1. Figure 1a shows the central connector: the conceptualization arrow. The rule states that a PP can be the agent of a primitive ACT. 1b says that a PP can be modified by a PA. 1c shows that a PP can change from one state or relation to another. 1d states that an ACT can have a PP as an object, while 1e

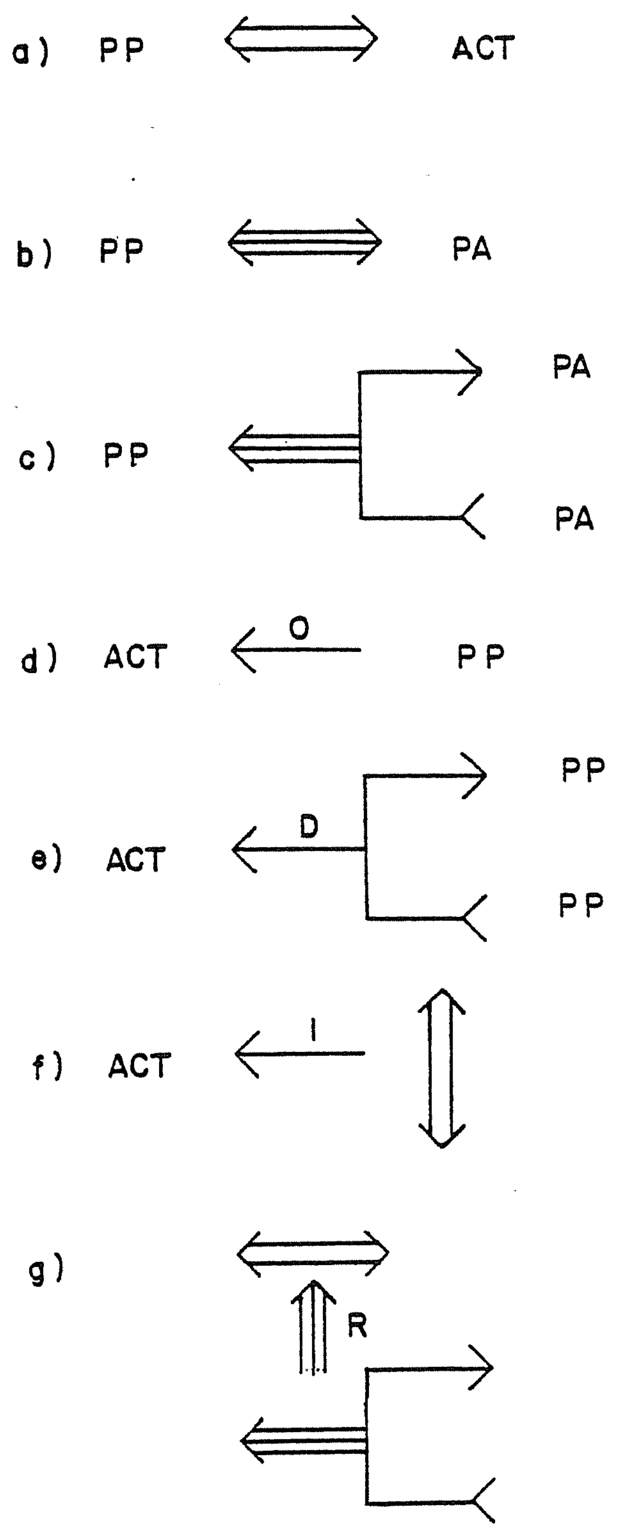


Figure 1. Conceptual Dependency connectors

shows that an ACT can have a direction from one location to another (a location is just a PP being treated as such). 1f illustrates the fact that an ACT can have an entire conceptualization as an instrument, while 1g says that a conceptualization can have some state or relation change as a result. There are a few more connectors in the CD system, but they are not necessary for understanding how Synner works. Those readers interested in a complete description should see (Schank, 1973) or (Schank, 1975).

The effect of all of these rules in constructing a CD net is difficult to see when presented individually. For an example of their use, see Figure 2, which illustrates one sense of the English verb "hit". This says that a hit involves a human propelling a physical object from himself toward another physical object, with the result that the two physical objects go from some unspecified relation to being in physical contact.

CD was chosen as the semantic representation for Synner primarily because of its high degree of language independence. Since it uses a small number of primitives rather than a large set of interconnected words from a natural language, it seems to assume less than other representations about the structure of any particular

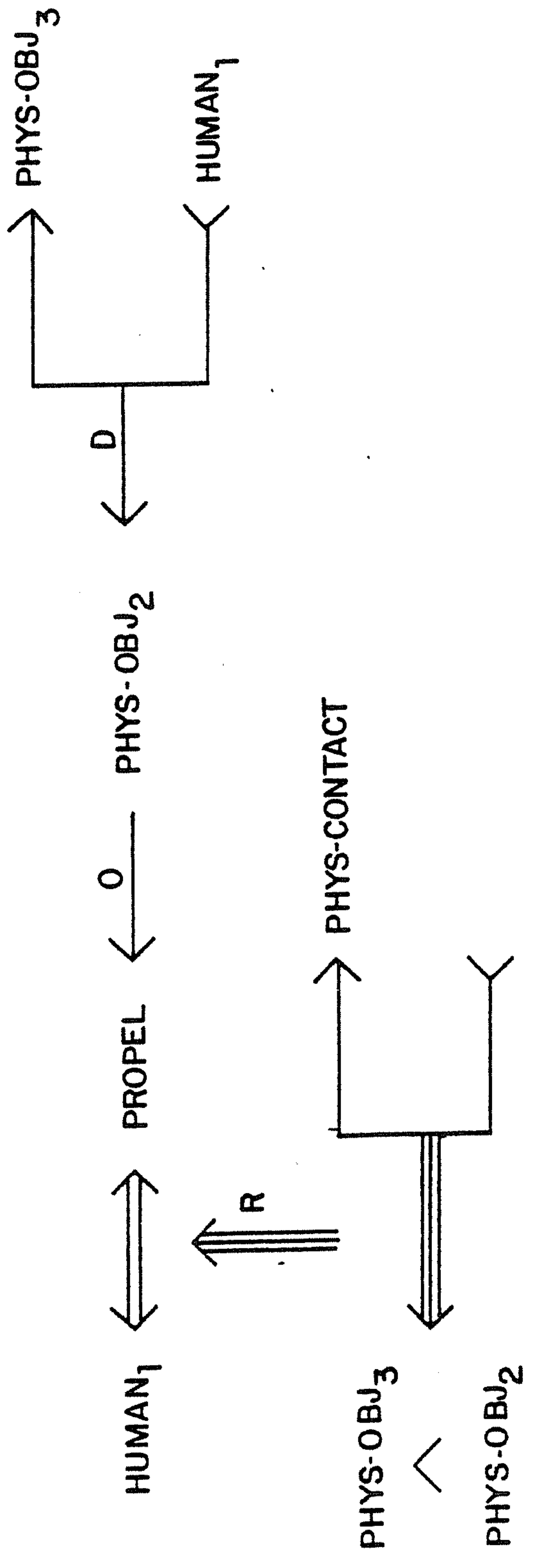


Figure 2. Conceptual Dependency structure

language. Because the primitives are based on more or less universal human concepts, CD should be as suitable for other human languages as it is for English. Furthermore, since Synner is modelling a real-world event through the use of a precoded semantic structure, the structure used should be one to which a perceptual analyzer could convert visual scenes. A system based on primitives seems to make this perceptual process more feasible than does a higher-level semantic representation that is closer to language.

Augmented Transition Networks

Augmented transition networks (ATNs) are widely used to represent the syntax of natural language in computers (Woods, 1969). They have the computational power of a Turing machine, which is necessary for grammars with the complexity of natural language. ATNs have the further advantage of being able to incorporate semantic knowledge.

ATNs are based on finite state machines (FSMs). Briefly, FSMs consist of several nodes (or states) connected by arcs and can be used to parse a string of symbols. There is a unique starting state and one or more final states. The arcs are labelled with symbols that can occur in the input stream. When at a given node, if the next symbol in the input labels an arc leading from that node, then that arc is traversed to its destination node and the symbol is deleted from the input stream. If the input symbol does not occur on a label for an arc leading from the current node, then the parse fails. If a final state is reached at the same time that the input stream is exhausted, then the input has been successfully parsed.

ATNs add several capabilities to the FSM formalism. First, as a notational convenience, it is not necessary to list input symbols on the arcs; instead, classes of symbols can be created and named, and only the class names need appear on the arcs. Second, ATNs can have subnetworks (or sub-machines). In place of input symbols or symbol classes, the name of a subnetwork can appear. In order to traverse the arc, a "push" to the named subnetwork is performed. That subnetwork operates on the input string starting from the current position in the string; if a final state in that network is reached, the parse is "popped" back to the calling level and the arc at that level is traversed. If the subnetwork cannot reach a final state, then the "pushing" arc is not traversed. Third, these subnetworks can be recursive; that is, a subnetwork can call itself either directly or indirectly, through a chain of other subnetworks. This recursiveness gives ATNs the power of context free grammars. Finally, ATNs have arbitrary registers which can be given values upon traversing an arc or tested for values before an arc can be traversed. This final augmentation of the FSM formalism gives ATNs the power of Turing machines. With this addition, the state of the parse is no longer fully defined by the current node together with the subnetwork call history; it is now also necessary to know the values of all the registers to specify the current state of the parse. General purpose programs could, in theory, be written as ATNs.

When representing the grammar of a natural language, ATNs are generally used as follows: the input symbols are words, and the registers are used to contain semantic information and to build a parse tree for the sentence.

In this thesis, the following conventions are used in presenting ATNs: 1) the parse always starts at S; 2) an arc labelled with the word "jump" means that the arc can be traversed without reading anything from the input string; 3) an arc labelled with "pop" indicates that the current network has been successful (i.e. reached a final state); 4) an arc labelled with a name alone indicates that a push to the subnetwork with that name is required; and 5) an "&" ("ϵ ") followed by a name requires a member of the named class of words to be present in the input sentence for the arc to be traversed.

For an example of the notation used, see Figure 3. This network would parse all and only the following four sentences:

"The boy hit the girl."

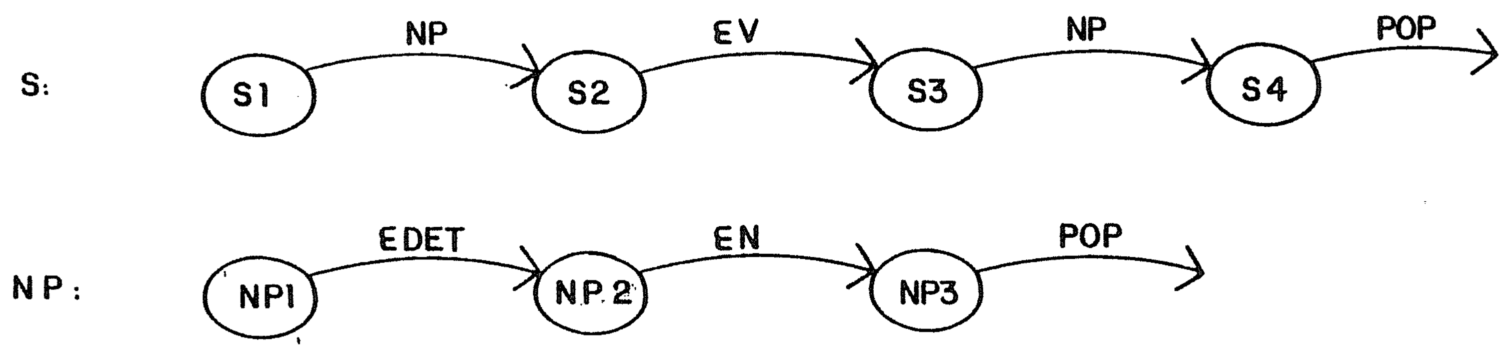
"The girl hit the boy."

"The boy hit the boy."

"The girl hit the girl."

A push is made to NP, which requires "the" followed by "boy" or "girl". If this is successful, then the parse is popped back to the top level and the arc is traversed to node S2. A "hit" is required next, followed by another push to NP. Thus, the four sentences above are allowed.

ATNs were chosen to represent Synner's grammar primarily for the following reasons: 1) they are well suited for implementation on a computer and 2) they have the power required to represent the syntax of natural language. Furthermore, there have been studies (Kaplan, 1972) that indicate that ATN grammars can handle well those language constructs that are easy for humans, while they have difficulty with some of the same constructs that cause problems for humans.



WORD	CLASSES
V	hit
DET	the
N	boy girl

Figure 3. Sample ATN

CHAPTER 3
DESCRIPTION OF SYNNER

Dictionary

The semantic structure of Synner's built-in dictionary is based on Schank's conceptual dependency representation. Synner's dictionary consists of four types of words: names of classes of physical objects (common nouns), names of individual physical objects (proper nouns), verbs, and states and relations between objects (adjectives).

A list of all the words in Synner's dictionary is presented in Appendix B.

1. Nouns

Objects are classified in a simple hierarchical tree structure (see Figure 4). The only exception to this strict hierarchy is the class of body parts, each of which can be related to several different classes by a part-of relation. For instance, head can be a part of man, boy, woman, etc. These relationships are indicated in each event structure where they apply rather than in the dictionary. An event showing John walking, for example, would involve movement of feet and legs, all of which would be indicated as part-of John. Figure 5 shows an event where HAND is part-of BOY-1.

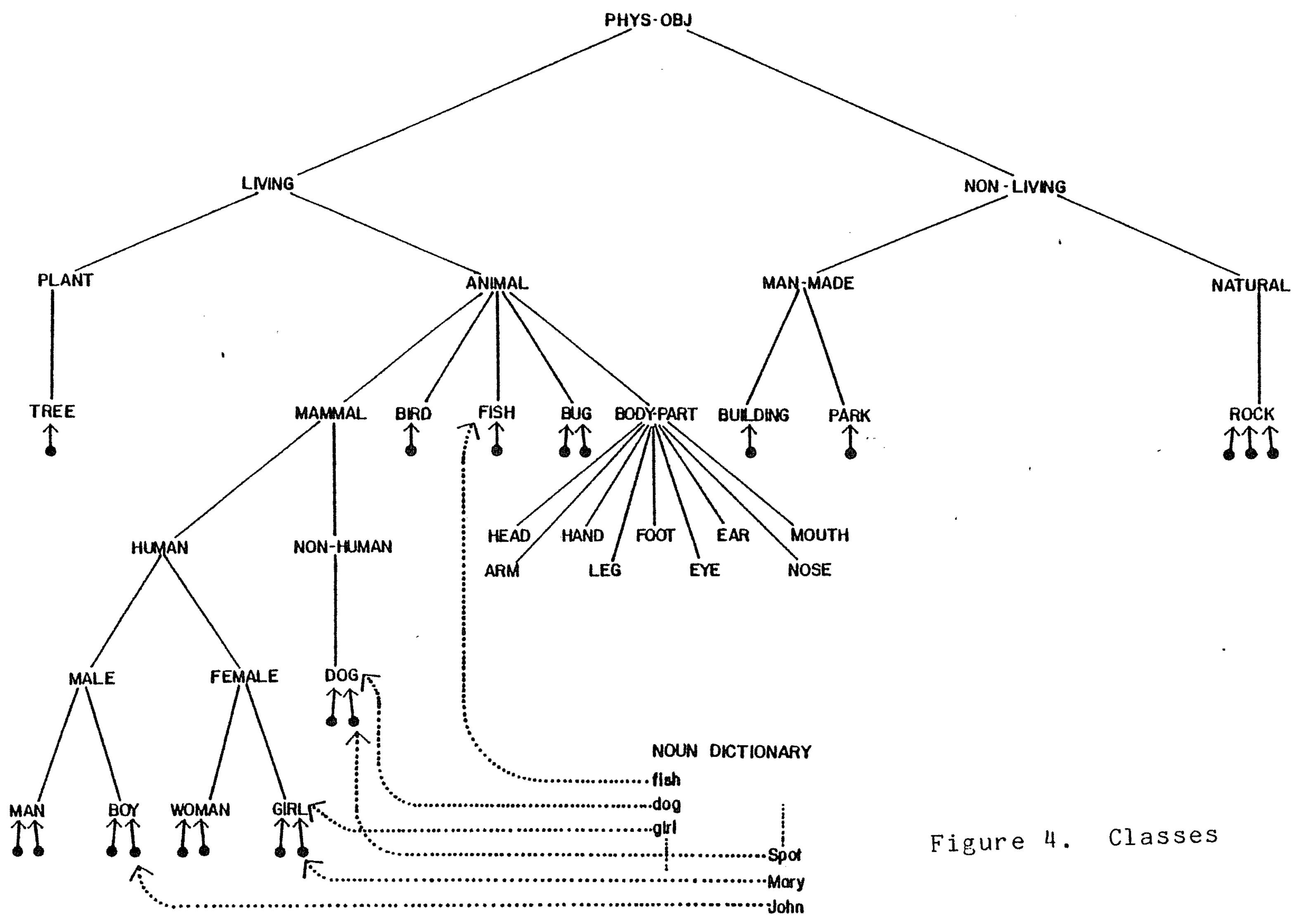


Figure 4. Classes

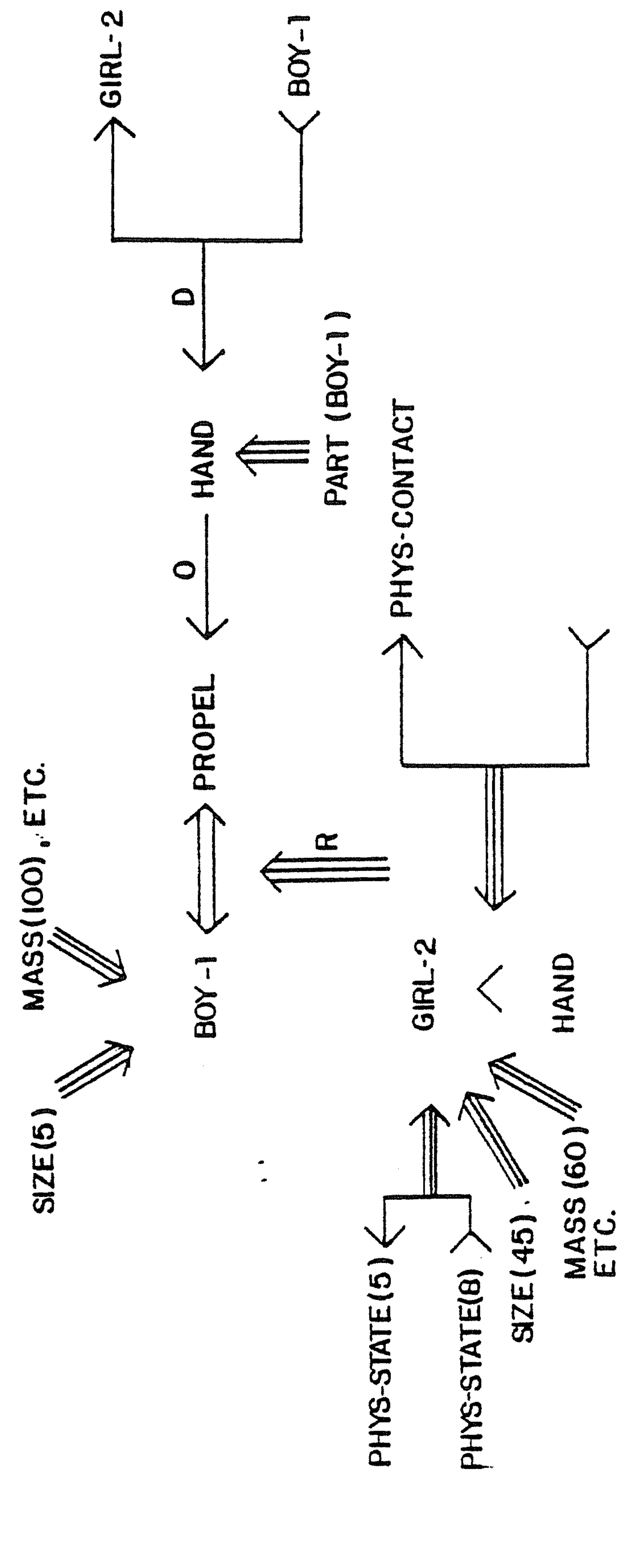


Figure 5. Typical event structure

At each leaf node of this tree (except body parts), there are one or more links to actual instances of that class. These are the "actual" physical objects that Synner knows about in the world.

Each class in the tree has one or more names attached to it from the dictionary. The instances of classes may or may not have names. For example, John and Mary are the names of two of the individual objects, but no rock has its own name.

2. Verbs

The definition of a verb is given as a Conceptual Dependency framework, where the picture producer slots are filled with class names, and state values are given as ranges of possible values. For a sample verb definition, see Figure 2 earlier in this thesis. The verb dictionary used by Synner is based upon and selected from one given by Schank (1973). Subscripts on the class names in the dictionary indicate whether the same or distinct objects must be used to fill the slots with two or more occurrences of the same class name. For example, every occurrence of PHYS-OBJ_2 must be filled by the same object, whereas PHYS-OBJ_3 must be a different object.

3. Adjectives

Adjectives are defined in terms of primitive states and relations, along with a range of possible values. For instance, "scared" is FEAR with a value in the range -7 to -2. One problem with this type of definition is the lack of context. "Small" is defined as MASS with a value less than 60. But what is small for an adult human might be average for a child and enormous for an insect. Thus, this means of definition has problems for a general knowledge structure. However, it does offer enough expressive power for the purpose of the learning that Synner is required to do.

Matching

The events that Synner observes are precoded in the same Schankian representation that is used in the dictionary. The difference is, where the dictionary has class names filling slots, the event has individual objects. A dictionary definition can be applied to any of a class of objects; but in a given event, some actual instance of the class is involved. See Figure 5 for a typical event.

When a sentence and event are presented to Synner, the first thing it must do is decide how the sentence is related to the event. This is done through reference to the dictionary.

First, the sentence is scanned for any words which are defined in the verb dictionary. When one is found, the event CD structure is compared with the dictionary definition for that verb. If a match is found, then words in the sentence which are defined in the noun and adjective dictionaries are compared to objects and states or relations in the event.

When this process is completed, Synner knows for each word in the sentence whether it has a referent in the event, and, if so, what that referent is and what its function is.

The following example may help clarify this process. The verb HIT is defined as in Figure 1. The event is like the one in Figure 5. A sentence describing this event might be, "The angry boy hit the scared little girl."

When the sentence is scanned for a verb, "hit" is found. The structure of the event is checked against the requirements for the verb HIT. As part of this matching, the objects involved in the event are compared to the classes in the dictionary entry for "hit". BOY-1 is found to be an instance of the class BOY, which is, through some intermediate classes, a subclass of HUMAN; thus, it meets the restrictions. For the second occurrence of $HUMAN_1$ in the dictionary, the same BOY-1 must match. If BOY-1 occurred in the event in that slot while BOY-2 filled the other slot, the match would fail.

After it is determined that "hit" does name the event that occurred, a search is made for nouns and adjectives. It is found that BOY-1 is an instance of the class BOY, and that the word "boy" names that class; thus, "boy" must name BOY-1 in the sentence. Similarly, "angry" names a state of BOY-1, "girl" names GIRL-2, "scared" names a state of GIRL-2, and "little" may name a state of the boy, the girl, or the rock (which exists in the event, but is not mentioned in the sentence). No match is found for the two occurrences of the word "the".

The match routine builds a list for each word in the sentence, describing whatever the matching process has been able to determine about the semantic role of that word in the sentence. These lists are then passed to the learning routine.

Learning

1. Initial ATN

After the matching of the sentence and event has been completed, it is necessary to use this information in building and modifying the grammar. There is a special case for the first sentence-event pair encountered: no previous ATN exists. Synner has a set of guidelines on how to go about constructing this first ATN, which is given below.

One way to create an ATN for a sentence would be to just run a linear string of arcs, one for each word, as in Figure 6. This grammar would always parse that sentence, but would not be readily generalizable. In order to get a more general grammar, Synner never puts word arcs in the main (top level) network; only pushes to subnetworks can occur on arcs in the top level net. Synner forms these subnetworks based on strings of contiguous words which are semantically related. Essentially, that means that each noun and each verb gets its own subnetwork. If an adjective is adjacent to the noun it modifies (as determined by the event match), then it is put in the same subnetwork with that noun; otherwise it gets a subnet of its own. Words like "the", which are not in the dictionary, are arbitrarily put into the same subnetwork with words that follow them in the sentence; unless there are no following words, in which case they are included with preceding words. Furthermore, an individual word is never placed on any arc, even in a subnetwork. Instead, a class of words is created (with just one member initially) and that word class name is put on the arc.

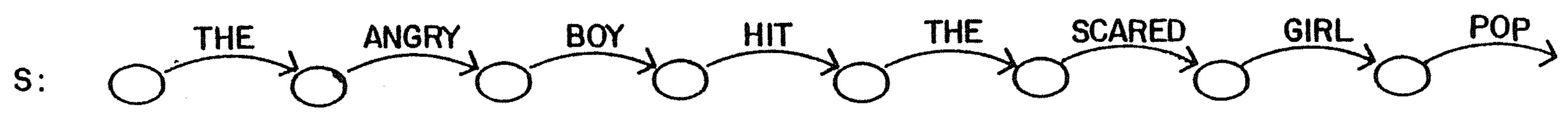


Figure 6. A not very useful ATN

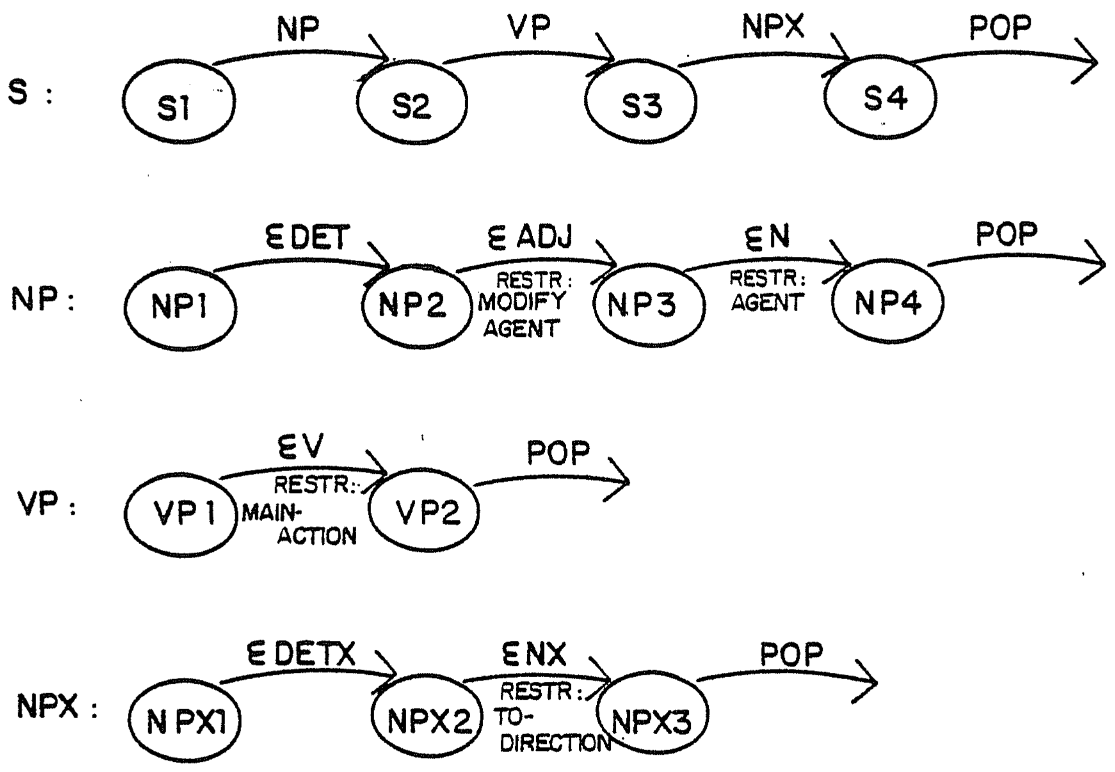
When these word class arcs are created, a restriction is placed on that arc indicating what semantic role a word must play in the sentence for that arc to be traversed. For nouns and adjectives, this semantic role is specified by what path was taken through the event network to reach the object or state named by that word. For verbs, there is simply an indicator as to whether it named the main conceptualization or a subordinate one. Words not in the dictionary, like "the", are put onto arcs with no semantic restrictions.

For the event in Figure 5, the sentence "The angry boy hit the girl", if the first sentence seen, would form the ATN shown in Figure 7. (The subnet and class names are mnemonic ones that have been substituted for the arbitrary ones Synner would use.)

2. Modifying ATNs

2.1. Finding differences

After a first ATN has been constructed, a different method of learning is used. The matching of the sentence with the event takes place as with the first ATN. But, then, an additional step is taken: the sentence with the word marked as to semantic role is parsed with the pre-existing ATN. This parsing does more than check for the



WORD CLASSES

DET	the
ADJ	angry
N	boy
V	hit
DETX	the
NX	girl

Figure 7. ATN for "The angry boy hit the girl."

grammaticality (according to the existing ATN) of the sentence -- it attempts to determine where and how this sentence differs from a legal sentence.

The search for differences starts with the smallest differences and works up to larger ones. The first thing that is tried is to allow a word that is not in the word class that the arc requires, as long as its semantic role is identical to one required for that arc. The second attempt is to loosen the word restrictions -- allow a match for nouns and adjectives if the beginning (first two elements) of the event path is the same, whether or not the word is identical.

As many of these word and restriction exceptions as necessary to parse are allowed in the sentence, and they may be combined with the higher level differences to be described below. Only one of any of the higher level differences is allowed in a sentence.

The first of the higher level parse relaxations to be tried is to skip a word arc; that is, traverse a word arc without swallowing a word from a sentence. This allows skipping a word that the grammar would otherwise require.

The next attempt at parsing is to delete any one word from the sentence. The effect of this is to permit a sentence which has a word that does not fit in the grammar.

The third of the higher level attempts is to allow any word to match a word arc; this is a more radical change than just slightly loosening the semantic restrictions.

Finally, there are the very high level differences. Again, only one of these is allowed per sentence, and they can only be combined with the two lowest level differences. The first of these is to allow any number of consecutive words from the sentence to be deleted, provided that the parse is at a point where it is about to push or has just completed a push to a subnetwork. The second is to allow any number of consecutive words to be deleted from the sentence, in place of a push to a subnetwork.

The parsing procedure constructs a parse path, noting what exceptions, if any, were allowed to the parse.

2.2. Modifying the grammar

All of the examples given in this section will be with reference to the grammar in Figure 7. The effect of the examples will not be cumulative; that is, each one will refer back to the original grammar in Figure 7.

If the parsing procedure found any exceptions to the ATN that, when allowed, resulted in a valid parse, the learning procedure uses these exceptions to modify the ATN. A summary of the rules to be discussed in this section is given in Table I.

The low-level exceptions cause simple changes. When a new word was allowed on an arc, that word is added to the word class. For example, the sentence "An angry boy hit the girl" will add the word "an" to the word class DET. When semantic restrictions were relaxed, the new semantic role is added to those allowable for the arc.

The higher level exceptions result in slightly more complex changes. If a word arc was skipped, then a new jump arc is added to the grammar connecting the same two nodes that the skipped arc connects. For example, the sentence "The boy hit the girl" requires that the &ADJ arc be skipped. Therefore, the NP subnetwork is modified as shown in Figure 8. When a word from a sentence was skipped at a point in the parse before node X of the ATN, then a new word arc is added in the following way: a new node Y is created; all arcs entering X are redirected to Y; then both a word arc incorporating the new word, and a jump arc are created and link Y to X. The sentence "The angry boy hit the scared girl" requires that the word "scared" in the sentence be skipped. Then the subnetwork NPX is modified as in Figure 9, and the new word class &ADJX is created and given the single member "scared". If a word was replaced by another word regardless of restric-

Learning Rules

EXCEPTION TO PARSE	MODIFICATION TO ATN
matches except word, same semantic role	add word to class
matches except for slightly different semantic role	expand semantic restrictions
skip a word arc	add jump arc
skip a word in sentence	add new node, insert word arc and jump arc
traverse word arc with word that does not match that arc	add word arc
skip consecutive words before or after push	if words related: new subnet, with push and jump added at pushing level otherwise: optional learning
skip consecutive words in place of push	if words related: new subnet as option to old subnet otherwise: optional learning
no parse successful	optional learning

[Optional learning means the teacher has the option to force a series of new subnets to be formed.]

Learning Rules

Table I

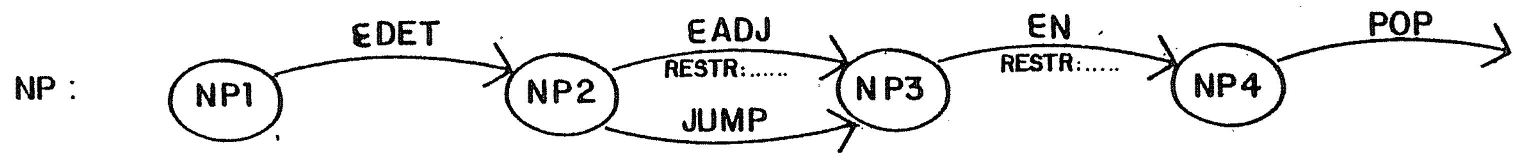


Figure 8. "The boy hit the girl."

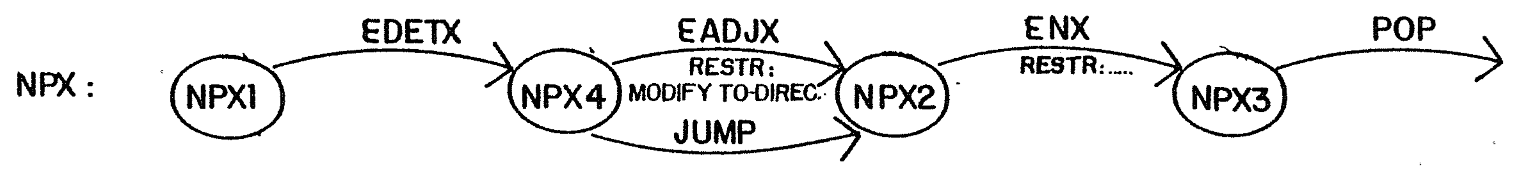
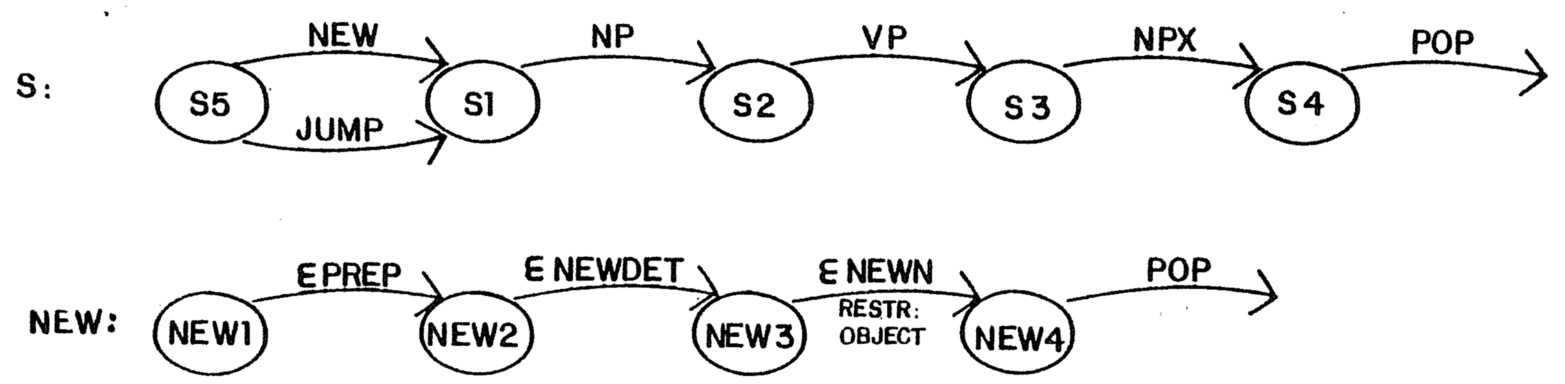


Figure 9. "The angry boy hit the scared girl."

tions, then a new word arc connecting the same two nodes as the replaced one is created.

The very high level exceptions can cause new subnets to be created. When a string of words from the sentence has been deleted before or after a push arc, then that string is checked to see whether all the words are closely related semantically. If they are, then a new subnetwork is formed, and a new node, a push arc and a jump arc are created at the pushing level, in much the same way that the new arcs and a node were added in the new word case. The sentence "With his hand, the angry boy hit the girl" requires skipping the first three words in order to be parsed. The words are found to be semantically related (by default, since "with" and "his" are unknown and will be related to anything), so a new subnet is created and the top level network is modified (see Figure 10). If the new JUMP is to a node that consists only of a POP, Synner replaces that JUMP with a POP. Thus, if the sentence were "The angry boy hit the girl with his hand," the effect on the top-level network would be as shown in Figure 11.

When a string of words is deleted in place of a push arc, that string is checked for semantic connectedness and made into a new subnetwork if it qualifies. A new push in parallel with the old push is added at the pushing level.



WORD CLASSES

⋮

PREP	with
NEWDET	his
NEWN	hand

Figure 10 "With his hand, the boy hit the girl."

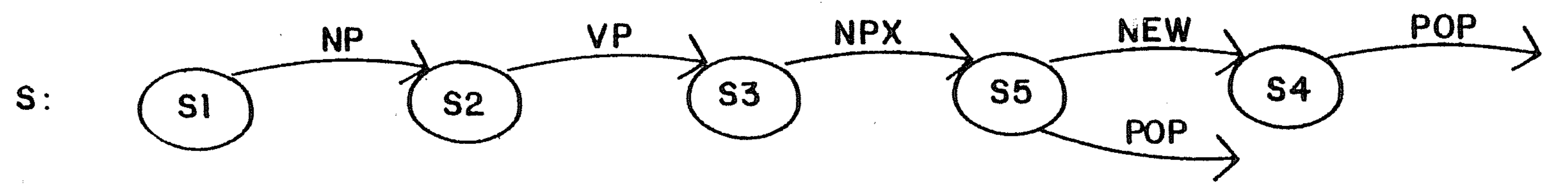


Figure 11 "The boy hit the girl with his hand."

The sentence "John hit the girl" would cause the grammar to be modified as shown in Figure 12. (If the skipped string was zero words long, a jump is added in place of the new push, thereby making that skipped network optional.)

The above description of when and how modification of the ATN is done still leaves some cases not accounted for. These cases are: the two different ways in which a string of consecutive words in the sentence is skipped, if the string was found not to be closely related semantically; and when the sentence did not parse at all.

There is an option that can be set by the user (teacher) to cover these cases. Most of the time this option is set so that no learning is done when one of the cases arises. The program simply prints a message that it could not sufficiently parse the sentence, and that nothing was learned. This ensures that no radical changes are made to the ATN; that changes are made a single step at a time.

However, it is occasionally desirable for Synner to learn a new type of sentence structure. The option can then be set so that learning is forced; it is as if the teacher were saying "Pay attention now. I'm going to show you something completely new." When this option is turned

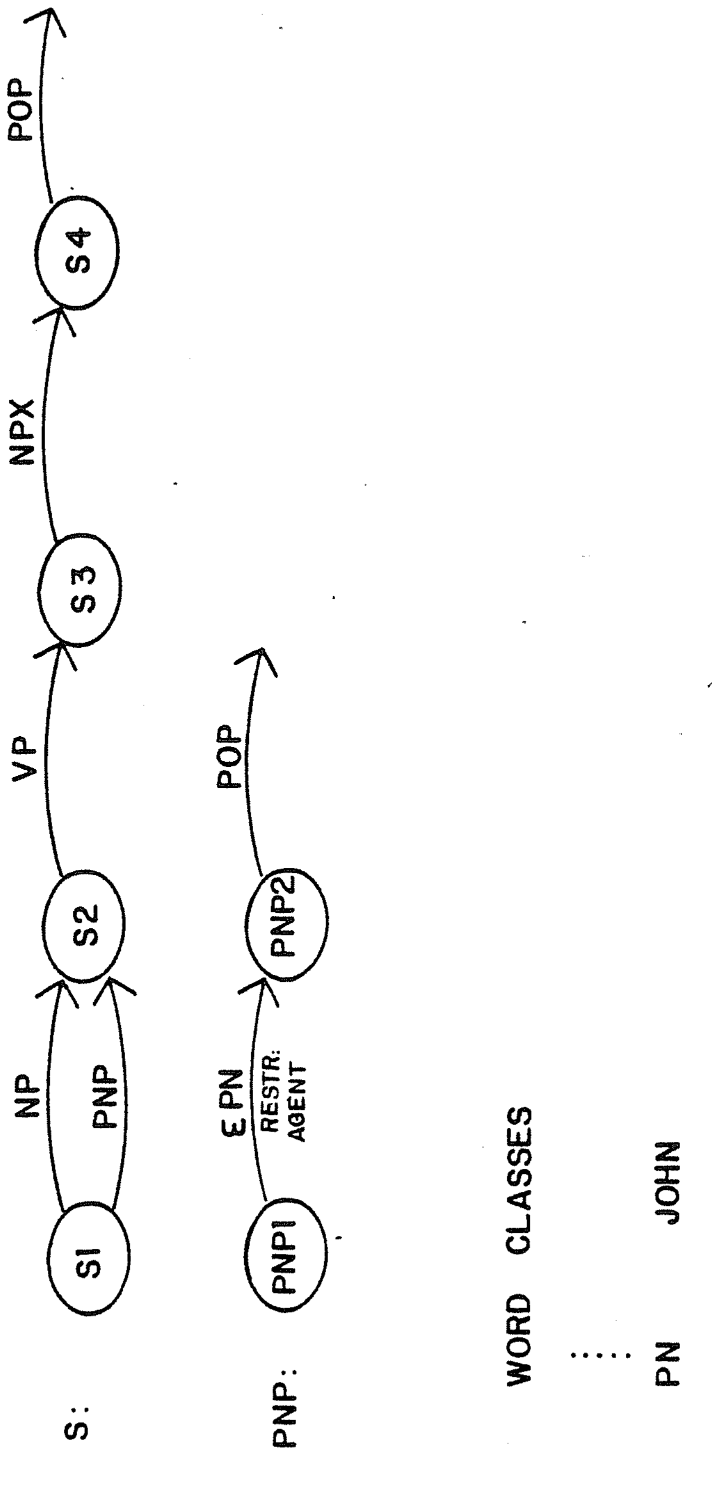


Figure 12. "John hit the girl."

on and one of the insufficient parses occurs, a process like that used for the first sentence that Synner sees is used. The words that could not be parsed (possibly the entire sentence) are broken into groups of semantically related words, and a subnet formed for each group.

Generalization

All of the learning mechanisms discussed above result in a grammar which continually grows in order to handle new constructs. However, the ideal grammar for a given language certainly is not the largest one. Therefore, it is desirable to have some mechanisms for restructuring the grammar, finding common features that can be merged. Synner has a limited generalization capability which it carries out when the teacher periodically gives a command to generalize.

An alternative for deciding when to generalize would be for Synner to invoke generalization after every _n_ input sentences. The final grammar would not differ by much from that obtained with the method actually implemented: if generalizations can be made, they are; if not, they are not. The only difference is that structures that could be generalized at a certain point using one method might have to wait a while before being modified using the other method. The primary reason that Synner is implemented as it is, with the trainer being responsible for invoking generalization, is to ensure that the Generalize procedure is invoked after the last sentence of the training sequence.

In the current version of Synner, one type of generalization has been implemented: loop forming. The loop-forming procedure searches the grammar for the existence of structures that would allow the occurrence of zero to two or possibly more consecutive instances of words with the same semantic restriction. (In English, adjectives modifying a noun can have this form; i. e. there may be zero adjectives, one, two, or more.) Because of the way Synner builds ATNs, the above condition is satisfied only by a structure like the one in Figure 13. The loop-forming procedure looks for occurrences of nodes that have only two arcs leaving the node, one jump arc and one word arc, where both arcs have the same destination node. When a node meeting these conditions is found, the destination node is checked for the same condition plus an additional constraint: the semantic restriction on the word arc must be the same as that on the word arc of the previous node. If these conditions are met, then loops will be formed. Before doing so, though, a check is made to see whether three, four, or more consecutive nodes meeting the conditions exist. If they do, all will be

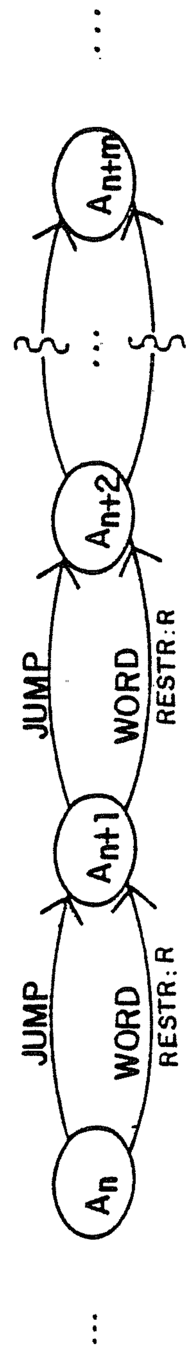
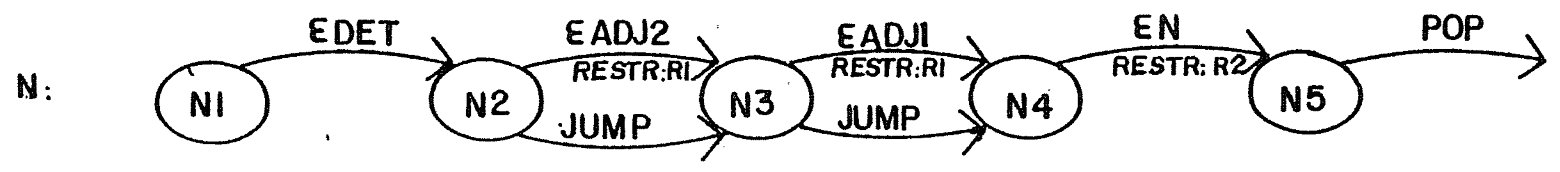


Figure 13. Loops can be formed.

merged into the loop.

To form the loops, the following procedure is followed: suppose A_b is the node that begins the repetitive sequence, A_e ends it, and A_{e-1} is the next to last. Then all the jump arcs in the sequence are deleted; the word arc entering A_e is modified so that it originates from A_e rather than A_{e-1} (thus looping from A_e to A_e); the words from the word classes referenced by the word arcs between A_b and A_{e-1} are merged into the new looping arc; all arcs pointing to node A_b are redirected to A_e; and nodes A_b to A_{e-1} are deleted. As an example, suppose we had the subnetwork in Figure 14. The loop-forming routine would find the conditions met and would modify the subnetwork, resulting in the subnet shown in Figure 15.

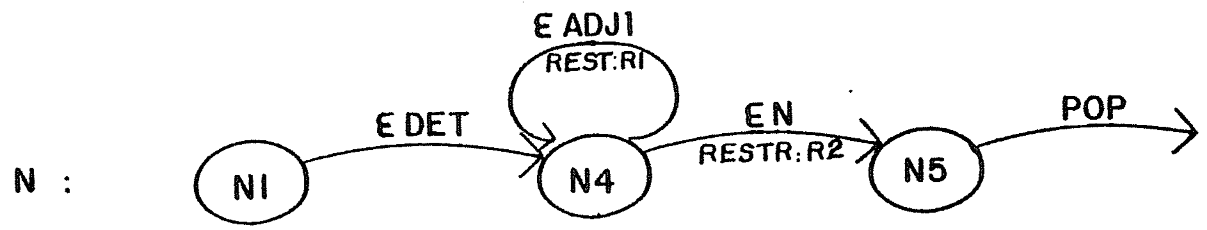
Other possible generalization techniques are discussed in the "Future Directions" section of Chapter 5.



WORD CLASSES

DET	the a
ADJ2	scared big
ADJ1	big angry
N	boy girl

Figure 14. Subnet before generalization.



WORD CLASSES

DET	the a
ADJ1	big angry scared
N	boy girl

Figure 15. Subnet after generalization.

CHAPTER 4

RESULTS

The program, Synner, was implemented in UW-Pascal on a Univac 1110 computer. It consists of approximately 6000 lines of code. The semantic memory and the augmented transition networks are both implemented as graphs of Pascal records. An outline of the structure of the program is given in Appendix A. Appendix C gives an example of how verbs and events are actually entered to and printed by the program. Appendix D has a sample learning step, showing all input and output.

Pascal was chosen as the implementation language for Synner because the record structures, pointers and user-defined typing available make it a straightforward matter to represent CD graphs and ATNs. The primary advantage of Pascal, though, is its strong type checking. The compiler together with its run-time support can perform a thorough check for misuse of variables, pointers, types, etc., thus making debugging easy and leading to confidence in the correctness of the final version of the program.

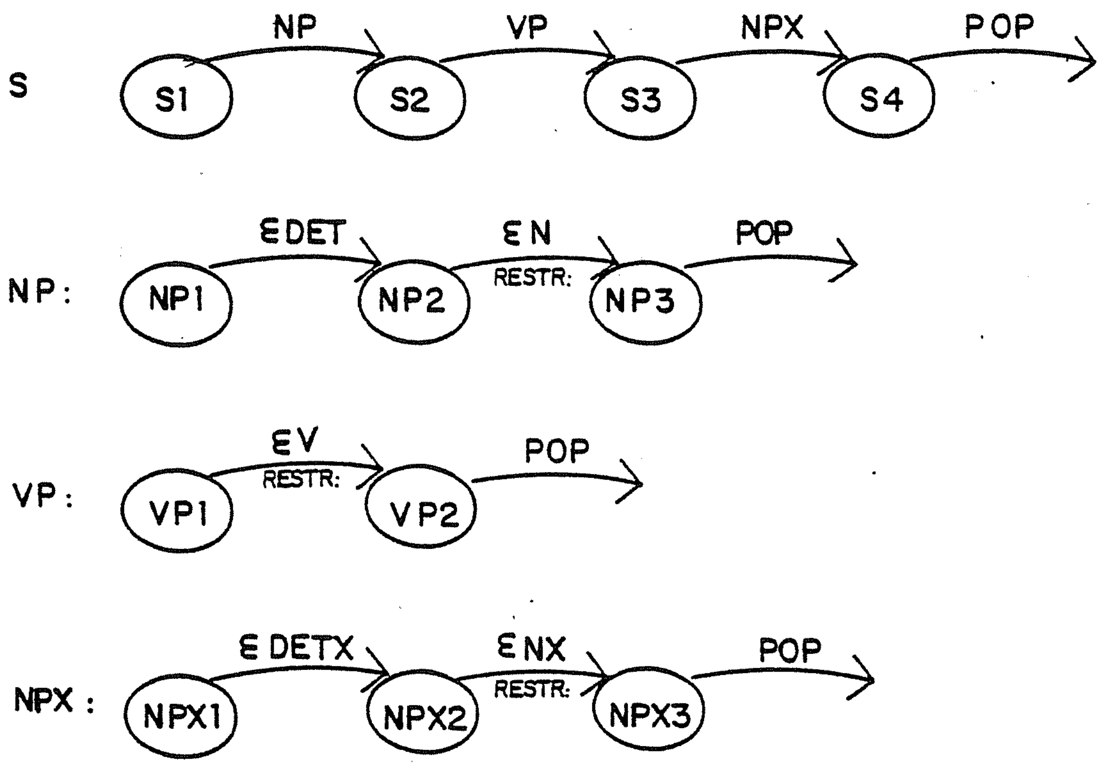
A sentence by sentence examination of how the learning mechanisms work in practice will now be undertaken. The teacher will first present simple sentences, and then gradually increase the complexity, building on previous knowledge.

First Sentence

At this point, Synner has no grammar at all. The teacher should start with a simple sentence that can be a basis for later sentences. The event is like the one in Figure 5, except that it shows a girl hitting a boy rather than the other way around. The sentence that is presented with this event is

"The girl hit the boy."

The sentence is compared to the event, and three word matches are found: "girl", "hit", and "boy". Since no previous grammar exists, the special rules for a first grammar are used. Since none of the words is closely enough related to any of the others, each is put into its own subnetwork. (For known words to be put into the same subnet, it is necessary that they all be nouns or adjectives that refer to the same object in the event.) The two occurrences of the unknown word "the" are arbitrarily put into the same subnetwork as the words they precede. The resulting grammar is shown in Figure 16. (After this, when a grammar is illustrated, only those parts of the grammar that have changed from the previously illustrated grammar will be shown.)



WORD CLASSES

DET	the
N	girl
V	hit
DETX	the
NX	boy

Figure 16. Grammar after sentence 1.

Second Sentence

The next event shows that boy hitting another girl with a small rock, and Synner is given the sentence

"The big boy hit the girl."

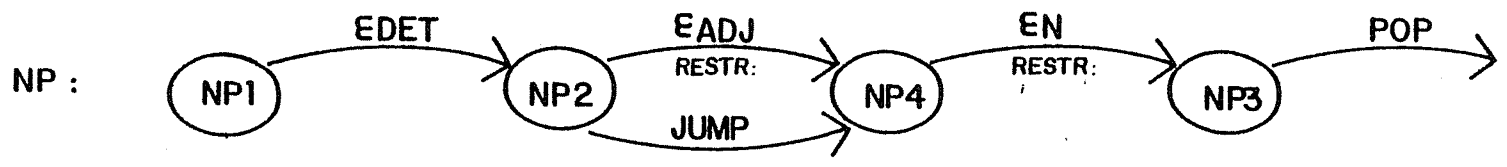
After matching to find the semantic role of each word, the existing ATN is applied to the sentence. The PARSE procedure discovers that if the word "big" were deleted and different words with the same semantic roles allowed in place of "girl" and "boy", then the sentence could be parsed. This information is passed along to the LEARN procedure. LEARN adds "boy" and "girl" to the N and NX word classes, respectively. Then, since "big" is closely related to "boy", a new word arc and a new jump arc are added to the NP subnetwork (see Figure 17).

Third Sentence

The same event is repeated, but this time the sentence

"A big angry boy hit the girl."

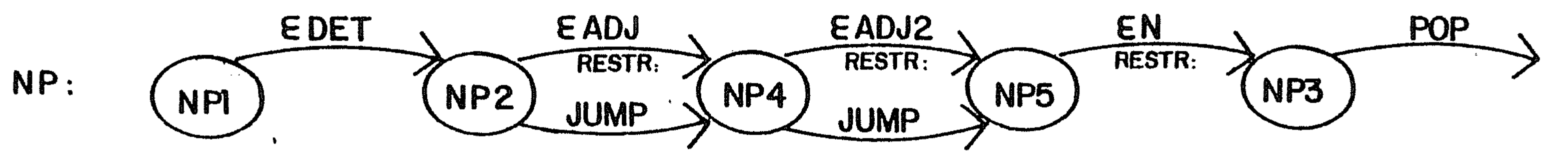
is presented. The PARSE routine finds that if "a" is substituted for "the" and the word "angry" deleted, the sentence can be parsed. LEARN uses this information to add "a" to the word class DET and to add another word arc and jump arc to subnet NP. The result is shown in Figure 18.



WORD CLASSES

DET	the
ADJ	big
N	boy girl
V	hit
DETX	the
NX	girl boy

Figure 17. Sentence 2.



WORD CLASSES

DET	a the
ADJ	big
ADJ2	angry
N	boy girl
⋮	⋮

Figure 18. Sentence 3.

Fourth Sentence

Now Synner is shown a girl hitting a boy again, along with the sentence

"The girl hit the boy with her hand."

This sentence can be parsed except for the three words at the end, so LEARN checks to see that they are semantically related. They are (by default, since nothing is known about "with" and "her"), so a new subnetwork PP is created and a push to PP and a pop arc are added to the main network (see Figure 19).

Fifth and Sixth Sentences

The next two sentences (presented together with appropriate events) are

"The boy hit the scared girl."

and

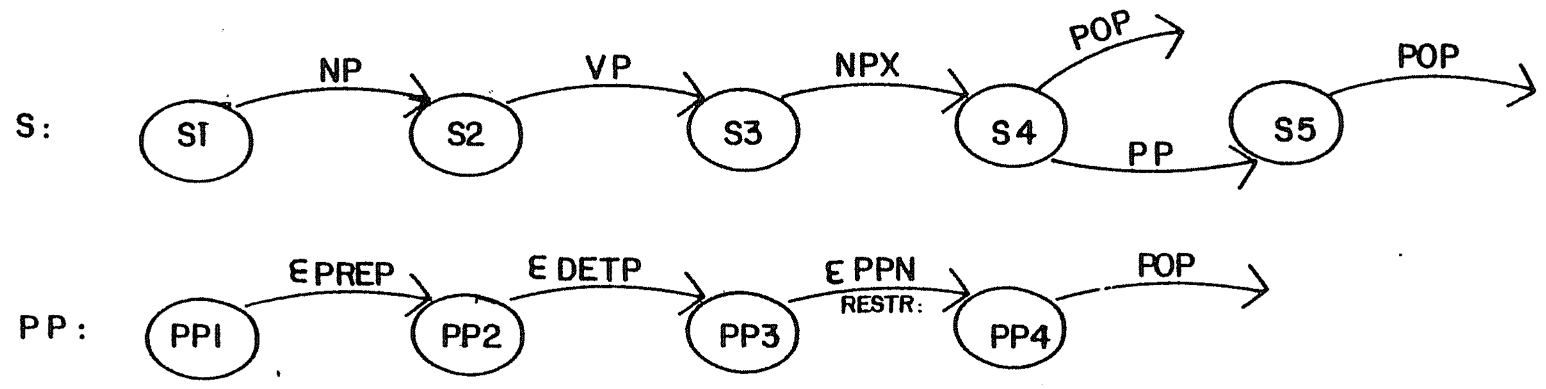
"The big boy hit the scared little girl."

These two sentences cause changes in the NPX subnetwork in much the same way that sentences 2 and 3 modified the NP network. Figures 20 and 21 show the grammars resulting from sentences 5 and 6, respectively.

Seventh Sentence

The seventh sentence presented is

"The big boy hit the little girl with a small rock."

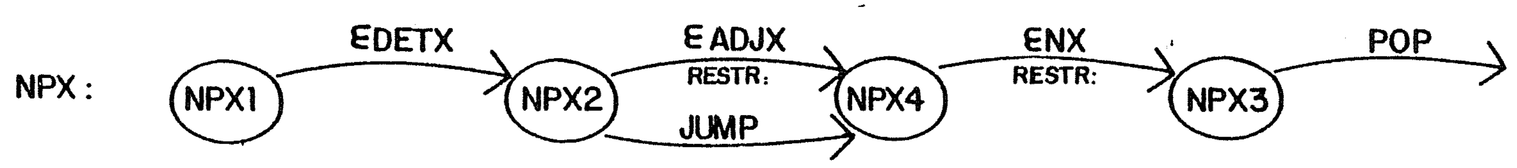


WORD CLASSES

⋮

PREP	with
DETP	her
PPN	hand

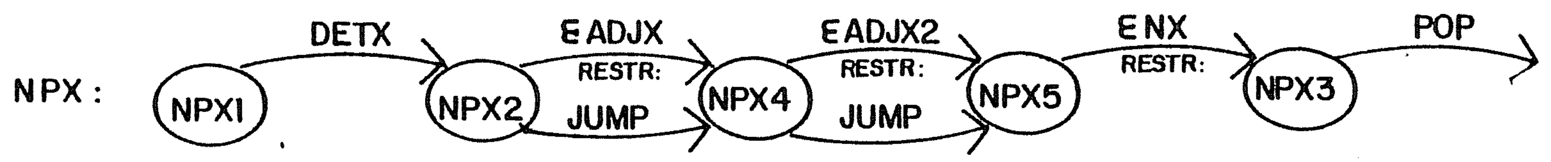
Figure 19. Sentence 4.



WORD CLASSES

DETX	a the
ADJX	scared
NX	girl boy

Figure 20. Sentence 5.



WORD CLASSES

⋮	
DET	a the
ADJX	scared
ADJX2	little
NX	girl boy
⋮	

Figure 21. Sentence 6.

An interesting feature of the sentence is that "little" and "small" can each refer to either "girl" or "rock". In the parsing phase, "little" will parse correctly only if it is related to "girl", so it is assumed that that is the case. "Small" is an excess word, so that information is passed to LEARN. LEARN finds that if it assumes that "small" is referring to "rock", it can insert a new arc into the PP subnet. It therefore makes that assumption and modifies PP. There is also another minor change -- adding "a" to DETP. The result is shown in Figure 22.

Generalization

At this point Synner is told to generalize. In both the NP and NPX subnets it finds structures which can turn into loops; each subnet allows zero, one or two occurrences of adjectives. These two subnetworks are modified and the corresponding word classes are merged.

The PP subnetwork is not modified, since it allows either zero or one adjectives, but not two or more; this is insufficient for the loop forming rules. The entire grammar resulting from this generalization is shown in Figure 23.

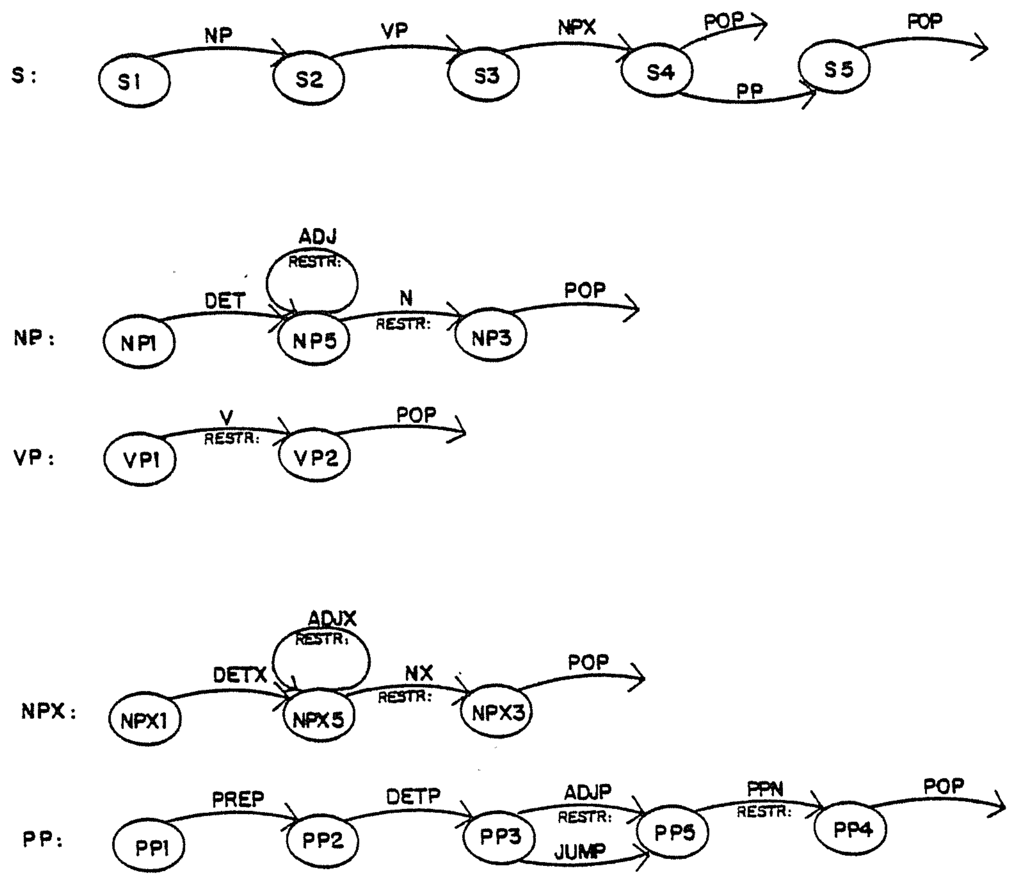
$$\begin{tikzcd}
\text{PP:} & \text{PP1} \arrow[r, "\epsilon\text{ PREP}"] & \text{PP2} \arrow[r, "\epsilon\text{ DETP}"] & \text{PP3} \arrow[r, bend left, "\epsilon\text{ ADJP}", "\text{RESTR:}"'] \arrow[r, bend right, "\text{JUMP}"] & \text{PP5} \arrow[r, "\epsilon\text{ PPN}", "\text{RESTR:}"'] & \text{PP4} \arrow[r, "\text{POP}"] & {}
\end{tikzcd}$$

WORD CLASSES

⋮

PREP	with
DETP	a her
ADJP	small
PPN	rock hand

Figure 22. Sentence 7.



WORD CLASSES

DET	a the
ADJ	angry big
N	boy girl
V	hit
DETX	a the
ADJX	little scared
NX	girl boy
PREP	with
DETP	a her
ADJP	small
PPN	rock hand

Figure 23. Grammar after generalization.

Eighth to Tenth Sentences

Sentence 8 introduces something new:

"John hit the little girl."

This sentence can be parsed only if the NP subnet is skipped and the word "John" is also skipped. Therefore, LEARN creates a new subnet, PNP, as an alternative to NP (see Figure 24).

The ninth sentence,

"Mary hit John." ,

does the same thing for NPX, creating PNPX as an alternative. It also adds "Mary" to the PN word class, as shown in Figure 25.

The tenth sentence,

"The boy hit Sue." ,

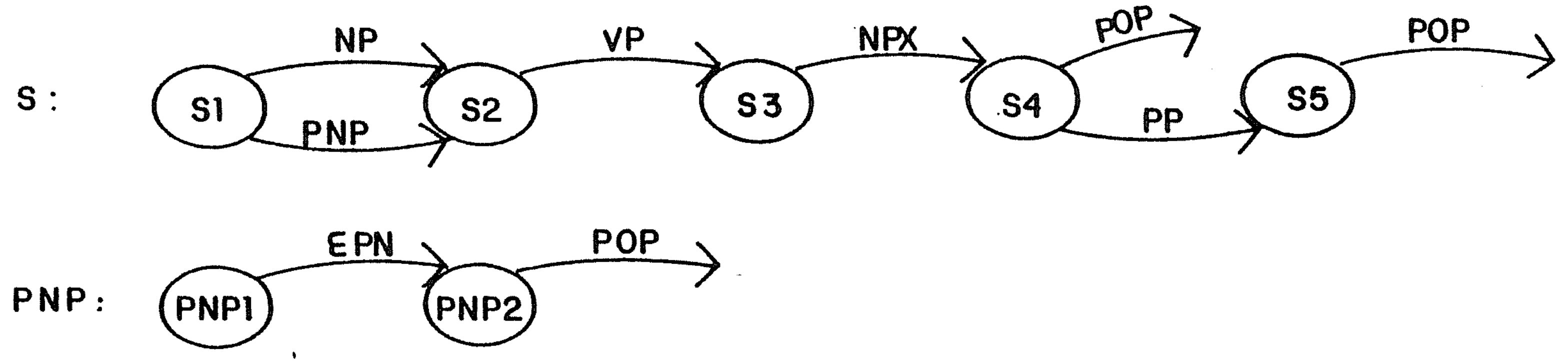
merely adds the word "Sue" to the PNX class.

Eleventh Sentence

The next sentence that Synner sees introduces the passive construction:

"The girl was hit by the boy."

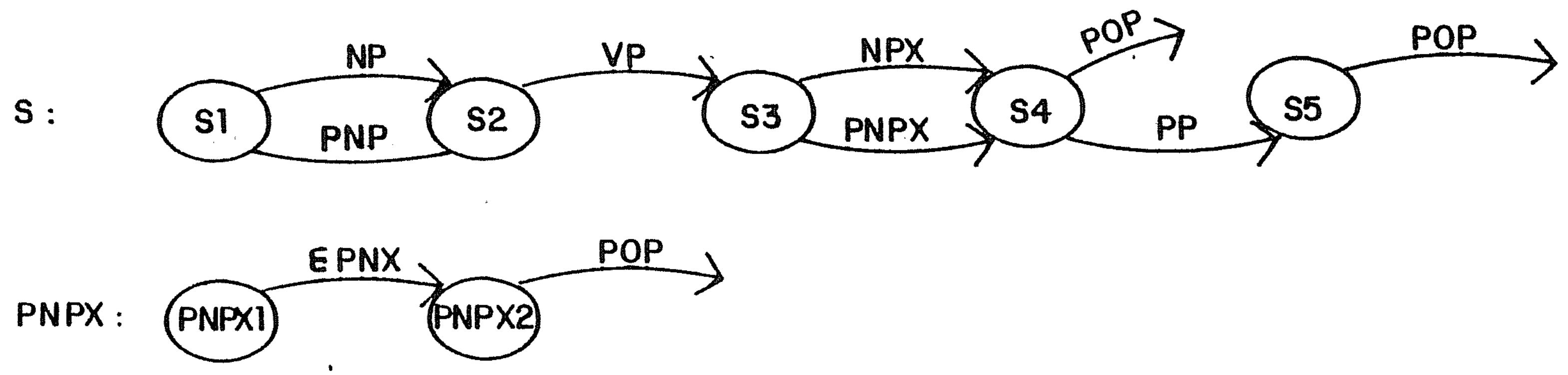
This sentence is so radically different from those that the grammar allows that it cannot be parsed using any of the PARSE procedure's rules. Synner therefore reports that it cannot learn anything from the sentence.



WORD CLASSES

PN	John

Figure 24. Sentence 8.



WORD CLASSES

PN	Mary	John
PNX	John	

Figure 25. Sentence 9.

The teacher then resets the option that tells Synner to create a grammar for this sentence, regardless of whether it can be parsed. The sentence is repeated, and since Synner still cannot parse it, an entirely new path through the top-level network is created, using the same rules as those used to build the first ATN. The new grammar is shown in Figure 26.

Twelfth to Fifteenth Sentences

These sentences merely duplicate for the passive voice what sentences 2, 3, 5 and 6 did for the active. The sentences are:

"The big boy was hit by the girl."

"The scared little girl was hit by the boy."

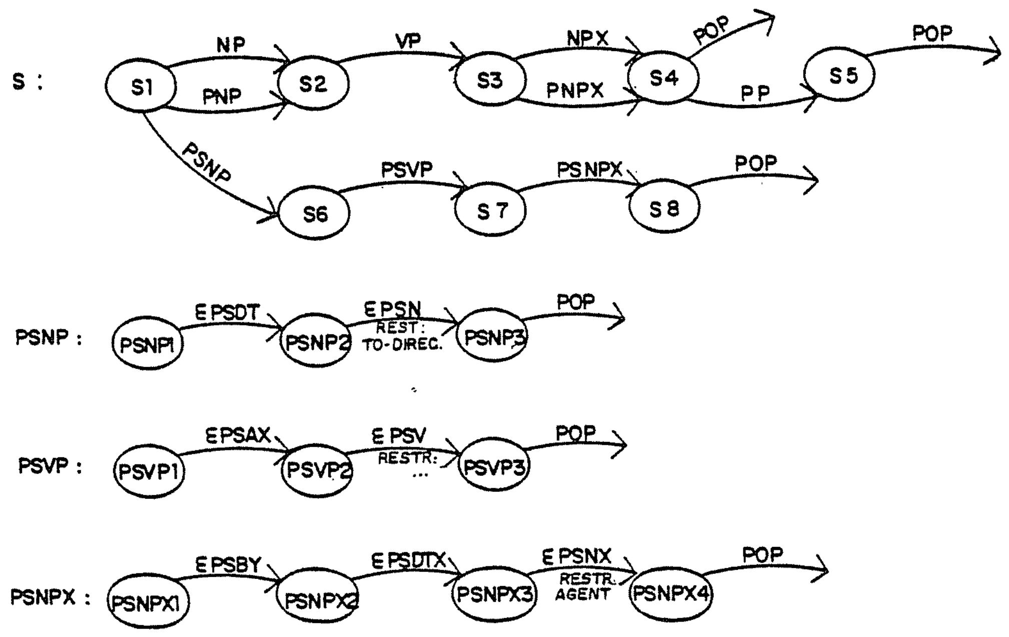
"The big surprised boy was hit by the angry girl."

"The girl was hit by the big angry boy."

At this point generalization is invoked, and adjective loops are formed for the passive voice just as they were for the active voice after sentence 7. The resulting grammar is shown in Figure 27. (The entire grammar in its raw, computer-printed form is shown in Appendix E.)

The Malevolent Teacher

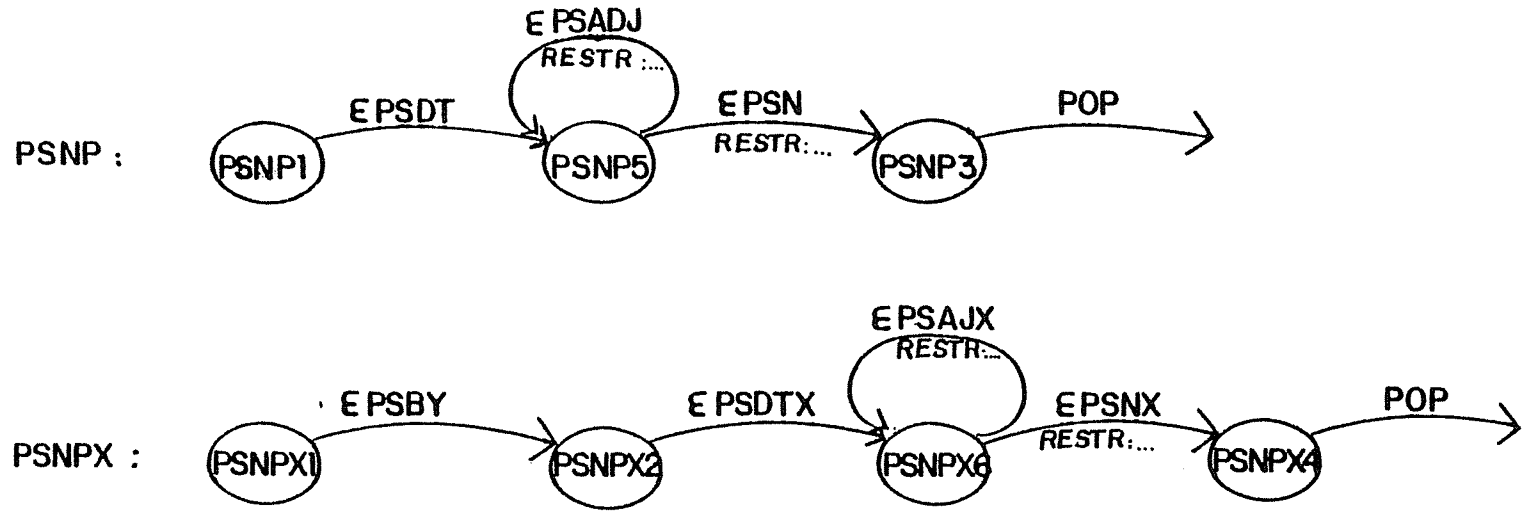
The sequence of sentences shown above was carefully chosen to build up the grammar one step at a time. The



WORD CLASSES

PSDT	the
PSN	girl
PSAX	was
PSV	hit
PSBY	by
PSDTX	the
PSNX	boy

Figure 26. Sentence 11.



WORD CLASSES

⋮

PSDT	the
PSN	boy girl
PSBY	by
PSDTX	the
PSNX	girl boy
PSADJ	surprised little scared big
PSAJX	angry big

Figure 27. Generalization after sentence 15.

result is that Synner was able to do a reasonable job of building a concise ATN. But the same set of sentences, presented in an order that is not as well formulated, could have far different results.

A new test sequence was run (starting with no grammar) that consisted of the same sentences as above but in the following order:
3, 1, 7, 4, 2, 6, 9, 8, 5, 10, 13, 15, 11, 12, 14, generalize.

This sequence was tried twice: once with the "brief" option on the entire time, so that no learning was done when the sentence was not parseable; and once with the "brief" option off, so that some learning was always forced. The first case resulted in a grammar that was larger than the grammar obtained from the well-planned sequence described above, since many times a new subnetwork was formed in place of an old one; the well-planned sequence would have produced just a word arc or jump arc. But, despite being larger, the grammar did not account for all the sentences, since several were not parseable at all. The second case resulted in a grammar that accounted for all the sentences, but was extremely large. Every time a sentence was not parseable, which occurred relatively frequently, an entire new path through the top-level network was created. In neither of the cases was the loop-forming mechanism useful; no one subnet allowed the occurrence of zero, one and two adjectives. Instead, these were spread across different subnets.

Appendix E shows the final grammars produced by these two runs.

CHAPTER 5
DISCUSSION

This chapter will discuss the limitations of Synner, some conclusions that can be drawn from the research, and some possible future directions for the research to take.

Limitations of Synner

Synner, as implemented, is limited to learning grammars of sentences involving action verbs with physical objects. This is not a limitation of the conceptual dependency (CD) formalism, which can represent (at least some) static states and abstract or mental actions. The situation that Synner is modelling is that of an event occurring along with a sentence describing that event. Since the event is presented to Synner precoded as a CD structure, abstract and mental actions could be included. But it is difficult to imagine a perceptual system on a computer that could observe thoughts being moved around in someone's brain or the abstract concept of transfer of ownership. Thus Synner has been limited to those actions for which a perceptual system is more feasible.

The use of registers in the ATNs that Synner constructs is limited to specifying the semantic role of a word on traversal of a word arc. This means that relationships among separated parts of a sentence cannot be represented in registers. This limitation implies that the grammars produced by Synner cannot have a complexity greater than that of a context free grammar (since the Turing Machine equivalence proof for ATNs depends on the existence of arbitrary registers).

Synner cannot currently learn a grammar with a subnetwork depth greater than two. That is, subnetworks of the top level network cannot in turn have subnetworks. This limits the complexity of the grammar that can be learned.

Whenever any generalizations are made by a learning program, whether something as simple as relaxing semantic restrictions or as complex as restructuring the network, it is possible for overgeneralizations to occur. Computer programs are not alone in this; human language learners also overgeneralize. For instance, children in general first learn the past tense forms of verbs on a case by case basis: go - went, etc. Later, after having learned several regular past tenses, they form a rule and apply it to all verbs. So where previously the correct form was used, they now use go - goed. After more time has passed and more examples have been heard, they relearn the irregular forms (Brown, 1973).

Synner does not yet have any automatic mechanisms for undoing overgeneralization. However, Synner's grammar is periodically checkpointed, that is, saved on external files. Thus, if the teacher notices an incorrect overgeneralization on the part of Synner, the grammar can be restored to a previous atate and training taken up from there. The training sequence is an important factor in determining whether and how often overgeneralization occurs. A well-designed training sequence (that builds slowly and carefully upon previous knowledge) can avoid much overgeneralization, while poorly designed ones could fool the program into making many incorrect generalizations.

Finally, the most serious constraint on the language for which Synner constructs a grammar: size. The size of the vocabulary is limited by machine storage considerations. The larger the vocabulary, the more expensive the program will become. A vocabulary approaching the size of a typical human vocabulary will be either impractical or impossible on present day computers. Each verb definition requires a rather large CD network, and the nouns must have an underlying network of classes. The current 86-word dictionary requires about 3500 36-bit words of storage on the Univac 1110.

Conclusions

The research presented here has indicated, by the existence of the program, Synner, that a relatively small set of heuristics can be used to generate an interesting grammar from events and sentences, even when the representation of the event is not closely related to English surface structure. Synner's MATCH procedure is the heart of the system. It finds the relationship of individual words in the sentence to the semantic structure of the event. It is this capability that is the primary feature distinguishing Synner from Anderson's LAS (Anderson, 1977). Because of this matching capability and the semantic representation that is not dependent on English structure as LAS's HAM representation is (LAS uses a bracketing procedure for sentences that depends on the relationship of the HAM memory structure to English surface structure), Synner is able to make fewer assumptions about the nature of the language to be learned and is still able to construct as good a grammar.

Another important factor demonstrated by the program is that the teacher plays a major role. A well designed training sequence that builds gradually on previous knowledge can result in steady, accurate learning. Poorly (or malevolently) designed training sequences can delay learning or even lead Synner down false paths. Most learning programs would have this same difficulty. If the examples presented to a program are misleading, it is difficult for the program not to be misled.

A third factor was discovered while testing Synner: the power of the learning rules should not be too great. Synner has a parameter that can be turned on to allow any two of its higher-level differences from the grammar to occur in one sentence. However, some tests made with this parameter turned on resulted in the grammar getting out of hand too easily. Incorrect generalizations occurred with alarming frequency. After this experience, all runs were made with this option turned off.

Future Directions

1. Near Future

The current version of Synner can learn grammars only for those sentences which contain action verbs; sentences that refer only to states, such as "the boy is angry", are ignored. The reason for this is that the MATCH procedure, which compares the sentence and event, requires the presence of an action verb in order to find the relationship between sentence and event. This procedure could be modified relatively easily to find the nouns and adjectives in a sentence even when no verb is present. The relationships that are found could then be used in building the grammar.

The tree structure of nouns connected by IS-A (i.e. subclass-of or instance-of) relations eliminates some interesting possibilities. This structure could be changed to a more general semantic net graph structure, with the possibility of a class being a subclass of more than one class, and has-as-part relations in addition to IS-A.

The current implementation of Synner uses ATN registers for just one purpose -- to specify the semantic role of a word found in conjunction with a word arc. The use of registers could be expanded, so that they could be set and tested in order to specify relationships among words and to specify the semantic role of an entire subnetwork. This would greatly expand the complexity of the language that could be learned. An example of an English construct that this would enable is "respectively". Consider the sentence

"Tom and Sue ate and drank brats and beer, respectively."

The current version of Synner could construct a grammar for this, but the grammar would not specify any constraints on which actor is performing which action on

which object. Sentences like
"Tom and Sue drank and ate brats and beer, respectively,"
or
"Tom and Sue drank and ate beer and brats, respectively,"
etc., would be accepted as valid descriptions of the same event described by the first sentence. With more liberal use of registers, the relationships among words in different parts of the sentence could be described and enforced.

There are other goals that a more general use of registers would enable, some of which will now be discussed. One type of generalization that Synner cannot yet make that Anderson's LAS can is the merger of two subnetworks into one. In LAS, whenever a phrase is successfully parsed by one subnetwork, a check is made as to whether any other subnetwork can parse that phrase. If it finds this compatibility often enough, the two subnetworks are merged into one. In Synner, since the word arcs specify what semantic role an individual word has, it is not very often that a subnetwork in one part of the grammar can be used to parse a phrase from a different part of the sentence. However, if registers were used to move the specification of the semantic role to the point of the call to the subnetwork rather than being within the subnetwork itself, then subnetworks could be merged even though different semantic roles are required by the different calls.

Currently, Synner forms only one level of subnetworks. The main network has subnetworks, but the subnetworks do not in turn have subnetworks. There is enough information available from the MATCH procedure about word relationships to enable a more complex structure. If words name entities that are in the same part of the conceptual dependency structure, then those more central to the main action could be put into the highest level subnetwork, those next most central and related to words in a given subnetwork could be put into a subnetwork of that subnetwork, and so on. All of this would require that registers be used to specify the relationships among the subnetworks. Implementing this would require defining another type of register in Synner's ATN formalism, require searching for partial matches of words with one another (based on their role in the event) instead of an all or nothing match, and require modifying the learning rules so that subnets are created when partial matches are found. In total, it would probably require approximately 400- 600 more lines of Pascal code.

Once subnetworks of subnetworks are possible and subnetworks can be merged, then recursive subnetworks become possible. Structures can be compared for repetition much as those that are turned into loops are compared. When the condition is met, the repetive parts can be replaced by a recursive call to the current subnetwork.

2. Longer Term Possibilities

At the present time, Synner does not have a stand-alone parser that can produce a CD structure given a sentence, nor a generator that can produce a sentence from a CD structure. The problems of parsing and generating sentences using ATNs have been studied by several researchers (Anderson, 1977; Schank, 1975), and Schank's students (Schank, 1975) have even done parsing and generating with ATNs using a conceptual dependency semantic representation. Therefore, no new theoretical ground would be broken by including them in Synner. However, their presence would enable someone unfamiliar with ATNs to judge the goodness of the induced grammar by examining the sentences that can be parsed and generated by that grammar.

A further use of a sentence generator would be to automatically check for overgeneralization. The current version of Synner requires manual intervention by the teacher to undo an overgeneralization by backing up to a previously checkpointed grammar. By using a sentence generator, every time a generalization was proposed Synner could produce several sentences dependent on the new feature of the grammar and ask for confirmation of their correctness. If the new feature resulted in ungrammatical sentences, it would not be incorporated into the grammar.

3. Distant Future

Synner has a built-in dictionary as its semantic basis, while Salveter's Moran (Salveter, 1979) is a system that learns concepts and verbs from pictures and sentences. If Synner were modified to use Salveter's conceptual memory structures (CMS) rather than Schank's conceptual dependencies as its semantic memory, then it would become feasible to merge Moran and Synner into a larger system. Moran would be building up a system of CMSs, while Synner would use those CMSs in its syntactic learning. This combined system would first learn concepts, then vocabulary, and finally syntax. The learning process would then continue with the conceptual memory, the vocabulary and the grammar all growing in parallel. This is closer to the way humans learn language than is either Moran or Synner individually. People do not stop learning vocabulary when they begin to learn grammar.

Summary

This thesis has discussed a computer program, Synner, that attempts to learn an augmented transition network (ATN) grammar for English. The process being modelled is that of a human learner who already has some conceptual knowledge and a vocabulary relating to that knowledge. This learner observes a real-world event and hears a sentence describing that event, and then adjusts his rules of grammar, if necessary, to account for the new sentence.

Synner's conceptual knowledge and vocabulary are predefined in terms of conceptual dependency (CD) theory, a representation based on a small number of primitives. To avoid the problems of perception, the "real-world" event is hand-coded into a CD structure, while the sentence is given as a string of words on an interactive computer terminal.

The learning procedure begins by comparing the sentence to the event using its built-in knowledge of the vocabulary. This matching process determines the semantic role of each word in the sentence. A parse of the sentence is then attempted using the current ATN grammar. There is a set of rules specifying how the sentence is allowed to differ from what is allowed by the grammar. The search for differences proceeds from the smallest ones

to larger ones, in order to insure that the least possible difference that can account for the sentence is found. The differences discovered, if any, are used in altering the ATN so that the new sentence is legal.

The ATN grammars obtained from Synner have shown that it is feasible for a computer program to infer rules of syntax of a natural language by comparing sentences to an internal encoding of the events they describe, even when that encoding is in the form of a CD network which is not specific to the structure of English. The conciseness and completeness of the grammar in representing the sentences that have been seen is heavily dependent on the order of presentation of the sentences: sentences that build gradually on previous ones result in smaller and (generally) more complete grammar than when sentences are presented randomly.

The ATN networks that are learnable by the current version of Synner do not have a high level of complexity, but the techniques used and the results obtained show good promise toward developing a program with greater capabilities. An eventual merger with programs that learn other aspects of language would be a foundation for a more complete language learning system.

REFERENCES

Anderson, J.R. Language, memory, and thought. Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1976.

Anderson, J.R. Induction of augmented transition networks. Cognitive Science, 1977, 1, 125-157.

Anderson, J.R., & Bower, G.H. Human Associative Memory. Washington, D.C. : Winston, 1973.

Brown, Roger. A first language. Cambridge, Mass.: Harvard University Press, 1973.

Gold, E.M. Language identification in the limit. Information and Control, 1967, 10, 447-474.

Jordan, Sara R. Learning to use contextual patterns in language processing. Ph.D. thesis, Dept. of Computer Sciences, University of Wisconsin- Madison, 1972.

Kaplan, Ronald M. Augmented transition networks as psychological models of sentence comprehension. Artificial Intelligence, 1972, 3, 77-100.

Klein, S., Fabens, Herriot, Katke, Kuppin, Towster. The Autoling System. University of Wisconsin Computer Sciences Department Technical Report # 43, 1968.

Moeser, S.D. & Bregman, A.S. The role of reference in the acquisition of a miniature artificial language. Journal of Verbal Learning and Verbal Behavior, 1972, 11, 759-769.

Moeser, S.D. & Bregman, A.S. Imagery and language acquisition. Journal of Verbal Learning and Verbal Behavior, 1973, 12, 91-98.

Quillian, M.R. The teachable language comprehender: a simulation program and theory of language. Comunications of the ACM, 1969, 12, 459-476.

Richards, I.A., et al. Russian through Pictures, Book I. New York: Washington Square Press, 1961.

Salveter, Sharon C. Inferring Conceptual Graphs.

Cognitive Science, 1979, _3_, 141-166.

Schank, Roger C. _The fourteen primitive actions and their inferences_. Stanford Artificial Intelligence Laboratory Memo AIM-183, 1973.

Schank, Roger C. Identification of conceptualizations underlying natural language. In R. Schank and K. Colby (eds.), _Computer Models of Thought and Language_. San Francisco: W.H. Freeman and Company, 1973.

Schank, Roger C. _Conceptual Information Processing_. Amsterdam: North-Holland Publishing Company, 1975.

Siklossy, L. Natural language learning by computer. In H.A. Simon and L. Siklossy, _Representation and meaning : Experiments with information processing systems_. Englewood Cliffs, New Jersey: Prentice Hall,1972.

Uhr, Leonard. Pattern-String learning program. _Behavioral Science_, 1964, _9_, 258-270.

Woods, W.A. Transition network grammars for natural language analysis. _Communications of the ACM_, 1970, _13_, 591-606.

APPENDIX A
OUTLINE OF THE PROGRAM

This outline is given in terms of a stripped down Pascal program, with the following conventions:

a) reserved words and predefined types are in upper case;

b) procedure and function names have the first letter of each word in upper case;

c) all other user defined entities are in lower case;

d) the underscore character (_) is legal in names.

```
PROGRAM Synner;

VAR
   do_learn, brief_option : BOOLEAN;
   event : ^event_structure;
   sentence_length : 0 .. 30;


   PROCEDURE Get_Event;

   BEGIN
      Get_Token;
      IF token = k_generalize THEN
         Generalize
      ELSE IF token = k_print THEN
         Print_Grammar
      ELSE
         {Build event structure}
   END;


   PROCEDURE Init;

   BEGIN
      Init_Classes;  {create the tree of classes}
      Init_Things;  {init. the instances of phys. objs.}
      Init_Token_Table;  {tables needed for verb and
```

```
                            event input}
   Init_States;  {init. adjectives}
   Init_Verbs;
   Init_Words   {create & alphabetize the dictionary}
END;


PROCEDURE Get_Input;

BEGIN
   IF NOT Eof THEN
      BEGIN
         Get_Event;
         Get_Sentence
      END
END;


PROCEDURE Match;


   FUNCTION Match_Act (event, dict : ptr_to_act)
      : BOOLEAN;

   BEGIN
      IF dict = NIL THEN match_act := TRUE
      ELSE IF event = NIL THEN match_act := FALSE
      ELSE match_act :=
         Match_Conc(event^.conc_ar, dict^.conc_ar)AND
         Match_Obj (event^.obj_ar, dict^.obj_ar) AND
         Match_Dir (event^.dir_ar, dict^.dir_ar) AND
         Match_Rec (event^.rec_ar, dict^.rec_ar) AND
         Match_Ins (event^.ins_ar, dict^.ins_ar) AND
         Match_aa (event^.aa_ar, dict^.aa_ar)
   END; {Match_Act}


   FUNCTION Match-Conc (event, dict : ptr_to_conc_ar):
      BOOLEAN;

   BEGIN
      IF dict = NIL THEN match_conc := TRUE
      ELSE IF event = NIL THEN match_conc := FALSE
      ELSE match_conc :=
         Match_Time(event^.time, dict^.time)AND
         Match_State(event^.result, dict^.result)AND
         Match_Act(event^.reason, dict^.reason)AND
         Match_State(event^.enab_by, dict^.enab_by)AND
         Match_pp(event^.subj_pp, dict^.subj_pp_cl)AND
         Match_pp(event^.loc_pp, dict^.loc_pp_cl)AND
         Match_pp(event^.desc_pp, dict^.desc_pp_cl)
   END; {Match_Conc}


   {Other Match routines are similar to
    Match_Act and Match_Conc}



BEGIN {Match}
   IF event = NIL THEN
      do_learn := FALSE
   ELSE
      IF sentence_length > 0 THEN
         BEGIN
            do_learn := TRUE;
            {Find the words in sentence defined in the
            verb dictionary, and then:}
            matched := Match_Act (event_ptr,
               verb[found].act)
            {Compare event with sentence,
            finding relationship}
         END
      ELSE
         do_learn := FALSE
END;


FUNCTION Parse : BOOLEAN;
   {Function Parse attempts to parse sentence using
    current grammar.  If it cannot, it tries to deter-
    mine where, why, and by how much the parse failed.
    Returns TRUE if it can determine this,
    FALSE otherwise}

TYPE
   matching_type = (m_none, m_exact, m_x_word, m_restr,
      m_arc_sk, m_wrd_del, m_arc_repl, m_new_push,
      m_push_sk);

VAR
   match_type : matching_type;
   cur_sent_pos : sent_length;
   repeating, new_choice : BOOLEAN;


   FUNCTION Try_Node (which_node:node_ptr) : BOOLEAN;
      FORWARD;
```

```
FUNCTION Try_Net : BOOLEAN;

BEGIN
   Try_Node ({first node})
END; {Try_Net}


FUNCTION Try_Word_Arc : BOOLEAN;

VAR
   matched : BOOLEAN;

BEGIN
   IF {current word meets restrictions} THEN
      BEGIN
         matched := TRUE;
         cur_sent_pos := cur_sent_pos + 1
      END;
   IF matched THEN
      matched := Try_Node ({next node});
   try_word_arc := matched
END; {Try_Word_Arc}


FUNCTION Try_Push_Arc : BOOLEAN;

VAR
   matched : BOOLEAN;

BEGIN
   matched := Try_Net;
   IF matched THEN
      matched := Try_Node ({next node});
   try_push_arc := matched
END; {Try_Push_Arc}


FUNCTION Try_Jump_Arc : BOOLEAN;

BEGIN
   try_jump_arc := Try_Node ({next node})
END; {Try_Jump_Arc}


FUNCTION Try_Node (which_node : node_ptr): BOOLEAN;

VAR
```

```
         matched : BOOLEAN;

      BEGIN
         matched := Try_Word_Arc;
         IF NOT matched THEN
            matched := Try_Jump_Arc;
         IF NOT matched THEN
            matched := Try_Push_Arc
      END; {Try_Node}


   BEGIN {Parse}
      match_type := m_none;
      REPEAT
         match_type := Succ (match_type)
         repeating := FALSE;
         cur_sent_pos := 0;
         REPEAT
            new_choice := FALSE;
            matched := Try_Net;
            IF cur_sent_pos <> sent_length THEN BEGIN
               repeating := TRUE;  matched := FALSE   END
         UNTIL matched OR NOT repeating
      UNTIL  matched OR (match_type = m_push_sk);
      parse := matched
   END; {Parse}


   PROCEDURE Learn;

   BEGIN
      IF do_learn THEN
         IF no_prior_learning THEN
            Create_New_Network
         ELSE IF Parse THEN
            Modify_Grammar
         ELSE IF brief_option THEN
            Writeln ('Could not parse.')
         ELSE
            Create_New_Network
   END;


BEGIN
   Init;
   Get_Saved_State;   {restore grammar saved
                        by previous run (if any)}
   WHILE NOT Eof DO
      BEGIN
```

```
         Get_Input;
         Match;
         Learn
      END;
   Save_State   {save grammar for next run}
END.
```

APPENDIX B

DICTIONARY

On the next page is a list of all the words that are defined in Synner's present dictionary. Following each word is an indication of what type of thing each word names: a Class of objects, an individual Thing, a state or relation (Adj), or a conceptual dependency structure (Verb).

1	ABSTRACTION	Class	44	HEAD	Class
2	AMAZED	Adj	45	HIT	Verb
3	ANGRY	Adj	46	HUMAN	Class
4	ANIMAL	Class	47	HUNGRY	Adj
5	ANYTHING	Class	48	JILL	Thing
6	ARM	Class	49	JOE	Thing
7	ARRIVE	Verb	50	JOHN	Thing
8	ARRIVED	Verb	51	LARGE	Adj
9	ASTOUNDED	Adj	52	LEG	Class
10	ATE	Verb	53	LITTLE	Adj
11	BIG	Adj	54	LIVING-THING	Class
12	BILL	Thing	55	LONG	Adj
13	BIRD	Class	56	MALE	Class
14	BOY	Class	57	MAMMAL	Class
15	BREAK	Verb	58	MAN	Class
16	BRING	Verb	59	MAN-MADE-OBJ	Class
17	BROKE	Verb	60	MARY	Thing
18	BROUGHT	Verb	61	MOUTH	Class
19	BUG	Class	62	NATURAL	Class
20	BUILDING	Class	63	NON-HUMAN	Class
21	CALM	Adj	64	NON-LIVING	Class
22	CAME	Verb	65	NOSE	Class
23	COME	Verb	66	PARK	Class
24	CONCRETE-OBJ	Class	67	PLANT	Class
25	DOG	Class	68	REPTILE	Class
26	DRANK	Verb	69	ROCK	Class
27	DRINK	Verb	70	SALLY	Thing
28	EAR	Class	71	SATIATED	Adj
29	EAT	Verb	72	SCARED	Adj
30	EYE	Class	73	SHOCKED	Adj
31	FEMALE	Class	74	SHORT	Adj
32	FISH	Class	75	SMALL	Adj
33	FOOT	Class	76	SPOT	Thing
34	FRIGHTENED	Adj	77	STRIKE	Verb
35	FURIOUS	Adj	78	SUE	Thing
36	GIRL	Class	79	SURPRISED	Adj
37	GO	Verb	80	TALL	Adj
38	GRAB	Verb	81	TERRIFIED	Adj
39	GRABBED	Verb	82	TREE	Class
40	HAND	Class	83	WALK	Verb
41	HAND	Verb	84	WALKED	Verb
42	HANDED	Verb	85	WENT	Verb
43	HARRY	Thing	86	WOMAN	Class

APPENDIX C

VERB AND EVENT INPUT/OUTPUT

This appendix illustrates how verbs and events are typed in for Synner to read and how they are printed out by Synner for humans to read. Verb input starts with the symbol "NEW" followed by quoted dictionary names for the verb. Each substructure in the CD network must be terminated with an "E" (for "end"). Each class name, since it may be modified by a list of state restrictions of arbitrary length, must be terminated with "epp" (for "end picture producer"). Event input is similar to verb input except that the structure is not named, things are used in place of classes, and the entry of each event is terminated with "endevt".

The output form of verbs and events is an attempt to approach as closely as possible the 2-dimensional graphic form of CD network representation.

Typical verb input:

```
NEW 'HIT'  ''   'STRIKE'  e
ACT  xpropel
  DEL SUBJ  human 1 epp  TIME 0
      RESULT  STATE  UNSPECRL  human 2 epp physobj 3 epp physcont E
   E
  OBJ  PHYSOBJ 3 epp
   E
  DIR  TO human 2 epp  E
E
```

Output form of same verb:

```
  1 HIT      1 STRIKE
        HUMAN  1<=>        XPROPEL
      Result
                 HUMAN  2 &      PHYS_OBJ  3  from     UNSPEC_REL   to      PHYS_CONT
        XPROPEL <-o-      PHYS_OBJ  3
        XPROPEL <-D-  to        HUMAN  2   from      NO_CLASS  0
```

Typical event input:

```
NEW
ACT  xpropel
  DBL SUBJ  girl1  anger -3 -3  fear -1 -1  mass 75 75  size 4.4 4.4 epp
      TIME 0
      RESULT  STATE  UNSPECRL  boy1 epp hand1 epp physcont E
   E
  OBJ  hand1  part hand1 epp  girl1 epp  e  epp  e
  DIR  TO boy1  surprise 5 5 physst 7 6  mass 101 101  size 5 5 epp  E
E  ENDEVT
```

Output form of same event:

```
   GIRL1    7 ( SIZE  4.4E+00  4.4E+00,  MASS  7.5E+01  7.5E+01,  FEAR -1 -1,  ANGER -3 -3, )
      <=>   XPROPEL
      Result
         BOY1     3 & HAND1   24  from   UNSPEC_REL  to   PHYS_CONT
    XPROPEL <-o- HAND1   24 (HAND1   ,GIRL1     PART:: PART, )
    XPROPEL <-D-  to   BOY1     3 ( SIZE  5.0E+00  5.0E+00,  MASS  1.0E+02  1.0E+02,
      PHYS_STATE  7  6,  SURPRISE  5  5, )   from  Unspc Th 0
```

APPENDIX D
SAMPLE OUTPUT

This appendix shows the printed output available from one cycle (that is, the processing of one sentence-event pair) of the program. (The cycle shown here is the second sentence discussed in chapter 4.) Part _a_ shows the event in its input form, and part _b_ illustrates how the event structure is printed by Synner. Part _c_ is the sentence that is given to Synner to describe the event. Part _d_ is debugging information that shows what part of the CD network each known word in the sentence names.

Part _e_ is also printed for debugging purposes; it is a coded format that shows how the sentence was parsed. "ND", "WD", "PU" and "JM" indicate whether the parse was at a node, a word arc, a push arc or a jump arc, respectively. The column after these symbols, if nonblank, indicates that there was an exception to the parse (see steps 5, 6 and 18). For example, "DL" indicates a word was deleted from the sentence, and "XW" means that the match was exact except for the actual word which filled the semantic role.

Part _f_ is the ATN grammar as it is printed by Synner. (This is the complete version of the grammar shown in Figure 17.) The subnet, node, and word class names are all arbitrary. An arc that does not say "push", "jump" or "pop" means that a member of the indicated word class is required. If an arc begins with "=" rather than "-", it indicates that there are semantic restrictions on traversing the arc. The word "again" in parentheses following a node means that the arcs leading from that node have already been printed.

Output from one step:

```
NEW
ACT  xpropel
  DBL SUBJ  boy1  anger -4 -4  fear -1 -1  mass 101 101  size 5 5 epp    TIME 0
      RESULT  STATE  UNSPECRL  girl2 epp rock1 epp physcont E
   E                                                                                  a
  OBJ  rock1   mass 10. 10. epp  e
  DIR  TO girl2  fear -6 -6 surprise 3 3 physst 6 3  mass 50 50  size 3.8 3.8 epp  E
E  ENDEVT

   BOY1     3 ( SIZE  5.0E+00  5.0E+00,  MASS  1.0E+02  1.0E+02,  FEAR -1 -1,  ANGER -4 -4, )
     <=>   XPROPEL
      Result
         GIRL2    8 & ROCK1   20  from   UNSPEC_REL  to   PHYS_CONT                      b
    XPROPEL <-o- ROCK1   20 ( MASS  1.0E+01  1.0E+01, )
    XPROPEL <-D-  to   GIRL2    8 ( SIZE  3.8E+00  3.8E+00,  MASS  5.0E+01  5.0E+01,
      PHYS_STATE  6  3,  SURPRISE  3  3,  FEAR -6 -6, )   from  Unspc Th 0

The angry boy hit the girl.                                                          c

All matches :
ANGRY           PAI  34                                                               d
BOY             CLS  34
HIT             VRB MAIN_VRB
GIRL            CLS  32

parse_string:
 1:   ND         a=1 |  2:   PU    w 3       |  3:   ND        a=1
 4:   WD    w 1      |  5:   ND DL w 1  a=1  |  6:   WD XW w 1
 7:   ND         pop |  8:   ND         a=1  |  9:   PU    w 1                        e
10:   ND         a=1 | 11:   WD    w 1       | 12:   ND        pop
13:   ND         a=1 | 14:   PU    w 2       | 15:   ND        a=1
16:   WD    w 1      | 17:   ND         a=1  | 18:   WD XW w 1
19:   ND         pop | 20:   ND         pop  |
OK
```

```
print

S              -->
  S01        -push-AAA -> S02      -push-AAB -> S03      -push-AAC -> S04      -pop->

AAA            -->
   AAA01     -AAA-a-----> AAA02    =AAA-c-----> AAA04    =AAA-b-----> AAA03    -pop->
                          AAA02    ---jump----> AAA04    (again)

AAB            -->
   AAB01     =AAB-a-----> AAB02    -pop->

AAC            -->
   AAC01     -AAC-a-----> AAC02    =AAC-b-----> AAC03    -pop->                               f

Word Classes:

AAA-a        :   THE
AAA-b        :   BOY           GIRL
AAB-a        :   HIT
AAC-a        :   THE
AAC-b        :   GIRL          BOY
AAA-c        :   ANGRY
_____________
```

APPENDIX E

ADDITIONAL GRAMMARS

This appendix shows the grammars produced by the sequence of sentences described in the "Malevolent Teacher" section of Chapter 4. The grammars are shown in their raw form as printed by the computer program. For comparison purposes, the final grammar resulting from the well-planned sequence of sentences is shown first.

Grammar resulting from good test sequence:

```
S          -->
  S01    -push-AAA-> S02    -push-AAB-> S03    -push-AAC-> S04    -pop->
                                                           S04    -push-AAD-> S05    -pop->
                                        S03    -push-AAF-> S04    (again)
  S01    -push-AAE-> S02    (again)
  S01    -push-AAG-> S06    -push-AAH-> S07    -push-AAI-> S08    -pop->

AAA        -->
  AAA01 --AAA-a---> AAA05 =-AAA-b---> AAA03 -pop->
                    AAA05 =-AAA-d---> AAA05 (again)

AAB        -->
  AAB01 =-AAB-a---> AAB02 -pop->

AAC        -->
  AAC01 --AAC-a---> AAC05 =-AAC-b---> AAC03 -pop->
                    AAC05 =-AAC-d---> AAC05 (again)

AAD        -->
  AAD01 --AAD-a---> AAD02 --AAD-b---> AAD03 =-AAD-d---> AAD05 =-AAD-c---> AAD04 -pop->
                                      AAD03 ---jump---> AAD05 (again)

AAE        -->
  AAE01 =-AAE-a---> AAE02 -pop->

AAF        -->
  AAF01 =-AAF-a---> AAF02 -pop->

AAG        -->
  AAG01 --AAG-a---> AAG05 =-AAG-b---> AAG03 -pop->
                    AAG05 =-AAG-d---> AAG05 (again)

AAH        -->
  AAH01 --AAH-a---> AAH02 =-AAH-b---> AAH03 -pop->

AAI        -->
  AAI01 --AAI-a---> AAI02 --AAI-b---> AAI06 =-AAI-c---> AAI04 -pop->
                                      AAI06 =-AAI-e---> AAI06 (again)
```

```
Word Classes:

AAA-a       :  A           THE
AAA-b       :  BOY         GIRL
AAB-a       :  HIT
AAC-a       :  A           THE
AAC-b       :  GIRL        BOY
AAA-d       :  ANGRY       BIG
AAD-a       :  WITH
AAD-b       :  A           HER
AAD-c       :  ROCK        HAND
AAC-d       :  LITTLE      SCARED
AAD-d       :  SMALL
AAE-a       :  MARY        JOHN
AAF-a       :  SUE         JOHN
AAG-a       :  THE
AAG-b       :  BOY         GIRL
AAH-a       :  WAS
AAH-b       :  HIT
AAI-a       :  BY
AAI-b       :  THE
AAI-c       :  GIRL        BOY
AAG-d       :  SURPRISED   LITTLE       SCARED       BIG
AAI-e       :  ANGRY       BIG
____________
```

Grammar for bad sequence with brief option:

```
S              -->
    S01    -push-AAA-> S02    -push-AAB-> S03    -push-AAC-> S04    -pop->
                                                             S04    -push-AAE-> S05    -pop->
                                          S03    -push-AAF-> S04    (again)
                                          S03    -push-AAG-> S04    (again)
    S01    -push-AAD-> S02    (again)
    S01    -push-AAH-> S06    -push-AAI-> S07    -push-AAJ-> S08    -pop->
                                          S07    -push-AAL-> S08    (again)
    S01    -push-AAK-> S06    (again)

AAA            -->
    AAA01  --AAA-a---> AAA02 =-AAA-b---> AAA03 =-AAA-c---> AAA04 =-AAA-d---> AAA05 -pop->

AAB            -->
    AAB01  =-AAB-a---> AAB02 -pop->

AAC            -->
    AAC01  --AAC-a---> AAC02 =-AAC-b---> AAC03 -pop->

AAD            -->
    AAD01  --AAD-a---> AAD02 =-AAD-c---> AAD04 =-AAD-b---> AAD03 -pop->
                       AAD02 ---jump---> AAD04 (again)

AAE            -->
    AAE01  --AAE-a---> AAE02 --AAE-b---> AAE03 =-AAE-c---> AAE04 -pop->

AAF            -->
    AAF01  --AAF-a---> AAF02 =-AAF-b---> AAF03 =-AAF-c---> AAF04 =-AAF-d---> AAF05 -pop->
                                         AAF03 ---jump---> AAF04 (again)

AAG            -->
    AAG01  =-AAG-a---> AAG02 -pop->

AAH            -->
    AAH01  --AAH-a---> AAH02 =-AAH-b---> AAH03 =-AAH-c---> AAH04 =-AAH-d---> AAH05 -pop->

AAI            -->
    AAI01  --AAI-a---> AAI02 =-AAI-b---> AAI03 -pop->
```

```
AAJ            -->
    AAJ01 --AAJ-a---> AAJ02 --AAJ-b---> AAJ03 =-AAJ-c---> AAJ04 -pop->

AAK            -->
    AAK01 --AAK-a---> AAK02 =-AAK-c---> AAK04 =-AAK-b---> AAK03 -pop->
                      AAK02 ---jump---> AAK04 (again)

AAL            -->
    AAL01 --AAL-a---> AAL02 --AAL-b---> AAL03 =-AAL-c---> AAL04 =-AAL-d---> AAL05 -pop->
```

Word Classes:

AAA-a	:	A	
AAA-b	:	BIG	
AAA-c	:	ANGRY	
AAA-d	:	BOY	
AAB-a	:	HIT	
AAC-a	:	THE	
AAC-b	:	BOY	GIRL
AAD-a	:	THE	
AAD-b	:	BOY	GIRL
AAE-a	:	WITH	
AAE-b	:	HER	
AAE-c	:	HAND	
AAD-c	:	BIG	
AAF-a	:	A	THE
AAF-b	:	SCARED	
AAF-c	:	LITTLE	
AAF-d	:	GIRL	
AAG-a	:	SUE	
AAH-a	:	THE	
AAH-b	:	BIG	SCARED
AAH-c	:	SURPRISED	LITTLE
AAH-d	:	BOY	GIRL
AAI-a	:	WAS	
AAI-b	:	HIT	
AAJ-a	:	BY	
AAJ-b	:	THE	
AAJ-c	:	GIRL	BOY
AAK-a	:	THE	
AAK-b	:	BOY	GIRL
AAK-c	:	BIG	
AAL-a	:	BY	
AAL-b	:	THE	
AAL-c	:	ANGRY	
AAL-d	:	GIRL	

Grammar for bad sequence without brief option:

```
S          -->
  S01    -push-AAA-> S02    -push-AAB-> S03    -push-AAC-> S04   -pop->
                                                          S04   -push-AAI-> S09    -pop->
                                        S03    -push-AAJ-> S04   (again)
                                        S03    -push-AAO-> S04   (again)
  S01    -push-AAD-> S02    (again)
  S01    -push-AAE-> S05    -push-AAF-> S06    -push-AAG-> S07   -push-AAH-> S08    -pop->
  S01    -push-AAK-> S10    -push-AAL-> S11    -push-AAM-> S12   -pop->
                                        S11    -push-AAN-> S12   (again)
  S01    -push-AAP-> S13    -push-AAQ-> S14    -push-AAR-> S15   -pop->
                                        S14    -push-AAW-> S15   (again)
  S01    -push-AAS-> S16    -push-AAT-> S17    -push-AAU-> S18   -pop->
                                        S17    -push-AAV-> S18   (again)

AAA        -->
  AAA01 --AAA-a---> AAA02 =-AAA-b---> AAA03 =-AAA-c---> AAA04 =-AAA-d---> AAA05 -pop->

AAB        -->
  AAB01 =-AAB-a---> AAB02 -pop->

AAC        -->
  AAC01 --AAC-a---> AAC02 =-AAC-c---> AAC04 =-AAC-b---> AAC03 -pop->
                    AAC02 ---jump---> AAC04 (again)

AAD        -->
  AAD01 --AAD-a---> AAD02 =-AAD-c---> AAD04 =-AAD-b---> AAD03 -pop->
                    AAD02 ---jump---> AAD04 (again)

AAE        -->
  AAE01 --AAE-a---> AAE02 =-AAE-b---> AAE03 =-AAE-c---> AAE04 -pop->

AAF        -->
  AAF01 =-AAF-a---> AAF02 -pop->

AAG        -->
  AAG01 --AAG-a---> AAG02 =-AAG-b---> AAG03 =-AAG-c---> AAG04 --AAG-d---> AAG05 --AAG-e--->
1 AAG06 =-AAG-f---> AAG07 -pop->

AAH        -->
  AAH01 =-AAH-a---> AAH02 -pop->
```

```
AAI          -->
    AAI01 --AAI-a---> AAI02 --AAI-b---> AAI03 =-AAI-c---> AAI04 -pop->

AAJ          -->
    AAJ01 --AAJ-a---> AAJ02 =-AAJ-b---> AAJ03 =-AAJ-c---> AAJ04 =-AAJ-d---> AAJ05 -pop->

AAK          -->
    AAK01 =-AAK-a---> AAK02 -pop->

AAL          -->
    AAL01 =-AAL-a---> AAL02 -pop->

AAM          -->
    AAM01 =-AAM-a---> AAM02 -pop->

AAN          -->
    AAN01 --AAN-a---> AAN02 =-AAN-b---> AAN03 =-AAN-c---> AAN04 -pop->

AAO          -->
    AAO01 =-AAO-a---> AAO02 -pop->

AAP          -->
    AAP01 --AAP-a---> AAP02 =-AAP-b---> AAP03 =-AAP-c---> AAP04 =-AAP-d---> AAP05 -pop->

AAQ          -->
    AAQ01 --AAQ-a---> AAQ02 =-AAQ-b---> AAQ03 -pop->

AAR          -->
    AAR01 --AAR-a---> AAR02 --AAR-b---> AAR03 =-AAR-c---> AAR04 -pop->

AAS          -->
    AAS01 --AAS-a---> AAS02 =-AAS-c---> AAS04 =-AAS-b---> AAS03 -pop->
                      AAS02 ---jump---> AAS04 (again)

AAT          -->
    AAT01 --AAT-a---> AAT02 =-AAT-b---> AAT03 -pop->

AAU          -->
    AAU01 --AAU-a---> AAU02 --AAU-b---> AAU03 =-AAU-c---> AAU04 =-AAU-d---> AAU05 =-AAU-e--->
1   AAU06 -pop->

AAV          -->
    AAV01 --AAV-a---> AAV02 --AAV-b---> AAV03 =-AAV-c---> AAV04 -pop->

AAW          -->
    AAW01 --AAW-a---> AAW02 --AAW-b---> AAW03 =-AAW-c---> AAW04 =-AAW-d---> AAW05 -pop->
```

Word Classes:

AAA-a	:	A	
AAA-b	:	BIG	
AAA-c	:	ANGRY	
AAA-d	:	BOY	
AAB-a	:	HIT	
AAC-a	:	A	THE
AAC-b	:	BOY	GIRL
AAD-a	:	THE	
AAD-b	:	BOY	GIRL
AAE-a	:	THE	
AAE-b	:	ANGRY	
AAE-c	:	BOY	
AAF-a	:	HIT	
AAG-a	:	THE	
AAG-b	:	LITTLE	
AAG-c	:	GIRL	
AAG-d	:	WITH	
AAG-e	:	A	
AAG-f	:	SMALL	
AAH-a	:	ROCK	
AAI-a	:	WITH	
AAI-b	:	HER	
AAI-c	:	HAND	
AAD-c	:	BIG	
AAJ-a	:	THE	
AAJ-b	:	SCARED	
AAJ-c	:	LITTLE	
AAJ-d	:	GIRL	
AAK-a	:	JOHN	MARY
AAL-a	:	HIT	
AAM-a	:	JOHN	

AAN-a	:	THE	
AAN-b	:	LITTLE	
AAN-c	:	GIRL	
AAC-c	:	SCARED	
AAO-a	:	SUE	
AAP-a	:	THE	
AAP-b	:	BIG	SCARED
AAP-c	:	SURPRISED	LITTLE
AAP-d	:	BOY	GIRL
AAQ-a	:	WAS	
AAQ-b	:	HIT	
AAR-a	:	EY	
AAR-b	:	THE	
AAR-c	:	BOY	
AAS-a	:	THE	
AAS-b	:	BOY	GIRL
AAT-a	:	WAS	
AAT-b	:	HIT	
AAU-a	:	BY	
AAU-b	:	THE	
AAU-c	:	BIG	
AAU-d	:	ANGRY	
AAU-e	:	BOY	
AAV-a	:	BY	
AAV-b	:	THE	
AAV-c	:	GIRL	BOY
AAS-c	:	BIG	
AAW-a	:	BY	
AAW-b	:	THE	
AAW-c	:	ANGRY	
AAW-d	:	GIRL	