Computing Story Trees

Alfred Correira

Department of Computer Science
The University of Texas at Austin
Austin, Texas 78712

A theory of understanding (parsing) texts as a process of collecting simple textual propositions into thematically and causally related units is described, based on the concept of macrostructures as proposed by Kintsch and van Dijk. These macrostructures are organized into tree hierarchies, and their interrelationships are described in rule-based story grammars related to the Kowalski logic based on Horn clauses. A procedure for constructing and synthesizing such trees from semantic network forms is detailed. The implementation of this procedure is capable of understanding and summarizing any story it can generate using the same basic control structure.

1. Introduction

One of the most difficult tasks in the field of computational linguistics is that of processing (parsing or understanding) bodies of connected textual material, from simple narratives like fairy tales and children's stories, to complex technical articles like textbooks and encyclopedia articles. When effective parsers were created capable of processing single sentences (Woods, 1970), (Schank, 1975b), (Norman and Rumelhart, 1975), (Winograd, 1972), it was quickly realized that these same techniques were not in themselves adequate for the larger task of processing sequences of sentences. The understanding of paragraphs involves more knowledge than and different knowledge from that necessary for sentences, and the structures produced by a text parser need not look like the structures of the sentences parsed individually.

However, the original impetus for current trends in text processing was the effort to solve problems of reference at the sentential level, in particular anaphora and ellipsis (Charniak, 1972). For example, in the paragraph

> John wanted to marry Mary. He asked her if she would marry him, but she refused. John threatened to foreclose the mortgage on the house where Mary's old sick father lived. They were married in June.

Simple-minded syntactic techniques are generally insufficient to resolve referents of the form of the "they" in the last sentence above. The human understander - and potentially the computer understander as well - requires real-world knowledge about threats, familial ties and marriage to realize that "they" refers to John and Mary.

Experiments with text processing led to such procedural constructs as frames (Minsky, 1975; Charniak and Wilks, 1976; Bobrow and Winograd, 1977), scripts and plans (Schank and Abelson, 1977), focus spaces (Grosz, 1977), and partitioned networks (Hendrix, 1976), among others. These efforts involved conceptual structures consisting of large, cognitively unified sets of propositions. They modelled understanding as a process of filling in or matching the slots in a particular structure with appropriate entities derived from input text.

There have also been rule-based approaches to the text processing problem, most notably the template/paraplate notion of Wilks (1975), and the story grammars of Rumelhart (1975). Although both approaches (procedures and rules) have their merits, it is a rule-based approach which will be presented here.

This paper describes a rule-based computational model for text comprehension, patterned after the theory of macrostructures proposed by Kintsch and van Dijk (1978). The rules are notationally and conceptually derived from the Horn clause, especially as described by Kowalski (1979). Each rule consists of sets of thematically, causally, or temporally related propositions. The rules are organized into a network with the macrostructures becoming more generalized approaching the root. The resulting structure, called the Story Tree, represents a set of textual structures.

Copyright 1980 by the Association for Computational Linguistics. Permission to copy without fee all or part of this material is granted provided that the copies are not made for direct commercial advantage and the *Journal* reference and this copyright notice are included on the first page. To copy otherwise, or to republish, requires a fee and/or specific permission.

0362-613X/80/030135-15$01.00

The process of generating from this network consists of choosing one of the rules to serve as the root of a particular story tree and recursively instantiating its descendants until terminal propositions are produced (Simmons and Correira, 1979). These propositions form the text of the generated story, requiring only a final phase to produce English sentences from the propositions. Conversely, a text is understood if its input sentences, parsed into propositions, can be mapped onto rules and these rules recursively mapped onto more abstract rules until a single node (the root) is achieved. Parsing and generating use the same rules in a similar manner for performing their respective tasks, and the rules lend themselves to a uniform tree structure possessing an inherent summarizing property.

2. Macrostructures and Story Grammar Rules

In this section the fundamental notion of macrostructure, as proposed and used by Kintsch and van Dijk, is presented and then analyzed from a computational, rather than a psychological, standpoint. An effective representation for macrostructures, derived from Horn clauses and organized into story trees, is described, as well as a data base for the representation.

2.1 Macrostructures

Kintsch and van Dijk (1975) present a system for organizing an entire discourse into a hierarchy of macrostructures, which are essentially metapropositions. The lowest level of a discourse textual representation is the set of input propositions that corresponds semantically to the text sentences, clauses and/or phrases. Propositions are conjoined by links of implication: if proposition A implies proposition B, then A and B are connected, and the link is marked with the strength of the connection, ranging from (barely) possible to (absolutely) necessary. The propositions and their connections reside in a text base. A text base can be either explicit, if all the implied information necessary for coherence is made explicit, or implicit, if propositions that can be assumed to be known or implied are omitted. A text is an explicit data base by itself, and all summaries of that text are implicit data bases. A college physics text would have a much more explicit text base than after-dinner conversation. The simple narrative texts examined in this paper have a text base between these two "extremes."

The sense in which "coherence" is used above is not defined precisely. Kintsch and van Dijk argue that coherence, or "semantic well-formedness", in a text requires, for each proposition in the text, that it be linked with one or more preceding propositions. This connection must exist for some reader in some context constrained by conventions for knowledge-sharing and assumption-sharing valid for that person in that context.

The result of this linking is a linear text base which is then mapped into a hierarchical structure in which propositions high in the structure are more likely to be recalled (via summaries) than those low in the structure. At the top of the hierarchy can be found propositions corresponding to rhetorical categories, such as "problem" and "solution," or narrative categories, such as "introduction," "complication," and "resolution."

Kintsch and van Dijk introduce a number of rules for relating these macrostructures to sets of input textual propositions: information reduction (generalization), deletion (of less important propositions), integration (combining events with their pre- and post-conditions), and construction (which relates complex propositions to their component sub-propositions).

There are two conditions that are always true regarding these macrostructures: a macrostructure must be implied by its subordinate propositions (i.e. encountering the subordinate propositions implies the existence of the macrostructure), and ordered sets of macrostructures collected together form a meaningful summary of the text. Kintsch and van Dijk believe that it is primarily macrostructures that are retained when a text is understood by a human reader and that the macrostructures are created as the text is being processed.

2.2 Macrostructures as Computational Constructs

As evidence in support of their theory, Kintsch and van Dijk present a number of psychological experiments in recall and summary with human subjects using as a text a 1600-word narrative taken from Boccaccio's *Decameron* (the Rufolo story). As a computational entity, a macrostructure is a node in a story tree whose immediate descendants consist of the subordinate propositions by which the node is implied, and is itself a descendant of the macrostructure it (partially) implies. Every macrostructure in this tree is the root of a derivation tree whose terminals are simple propositions.

Each level of the tree shares the attribute of summarizability, i.e. a summary of the text may be extracted from any level of the tree, becoming less specific as the summary level approaches the root. The lowest level summary is the original text itself; the highest level (the root) is a title for the text.

The ability to give meaningful (coherent) summaries for a text is one attribute of comprehension for that text, and any procedure yielding trees possessing the summary property can be said to partially understand the text. Consideration must also be given to classification schemas and rules for paraphrase, ana-

phora, and question-answering. Furthermore, given an appropriate data base internalizing the relationships between a macrostructure and its subordinate macrostructures or simple propositions (microstructures) and a summary derived from a story tree, it is possible for a procedure to reconstruct to a certain degree of detail the original text from which the tree was derived. The degree of detail recovered is directly dependent on the relative distance from the nodes forming the summary to the original input propositions (the leaves) of the text tree.

How is this subordinate relationship among propositions to be described formally to a computational process? One simple formulation is in the form of a rule

A <= B,C,D

meaning "you may assert the truth (presence) of macrostructure A if you can find the (nearly) contiguous propositions B, C, and D present in the input text." "Nearly" means that an allowable level of "noise," perhaps in the form of irrelevant side information, may be present between the specified propositions (a problem not addressed here).

This rule form closely resembles in structure and meaning the Horn clause notation. The general clause has the format

C[1],...,C[m] <= A[1],...,A[n]

where C[1],...,C[m] are elementary propositions forming the consequent, and A[1],...,A[n] are elementary propositions forming the antecedent. If the propositions in a clause contain the variables x[1],...,x[i], then the clause has the interpretation

for all x[1],...,x[i],
    A[1] and ... A[n] implies
    C[1] or ... C[m]

If the subscript m for a clause is zero or one, then that clause is referred to as a Horn clause. If m=1 and n=0, the Horn clause is called an *assertion*.

There are several differences between the Kowalski logic and the logic adopted here. One of these has to do with the ordering of the antecedent propositions. In a true Horn clause, the ordering is irrelevant and A <= B,C,D is as good a rule as A <= C,D,B, etc., i.e. the antecedents can be proved in any order. The ordering in the system described here is governed by rules of coherence. For example, the rule:

```
(TRADINGVOYAGE A RUFOLO WITH GOODS
     IN SHIP TO CYPRUS)
 <= (BUY A RUFOLO TH SHIP)
    (BUY A RUFOLO TH GOODS)
    (LOAD A RUFOLO TH SHIP WITH GOODS)
    (SAIL A RUFOLO TO CYPRUS MEANS SHIP)
```

is a meaningful rule. (Here case notation is used: A for Agent, TH for THeme, etc.) On the other hand,

```
(TRADINGVOYAGE A RUFOLO WITH GOODS
     IN SHIP TO CYPRUS)
 <= (SAIL A RUFOLO TO CYPRUS MEANS SHIP)
    (LOAD A RUFOLO TH SHIP WITH GOODS)
    (BUY A RUFOLO TH GOODS)
    (BUY A RUFOLO TH SHIP)
```

is nonsensical. The rules of coherence that order the antecedent propositions may involve several criteria: causal connectedness (B causes/is the result of C, which causes/is the result of D), or temporal ordering (B happens before/after C, which happens before/after D), etc. Note that the ordering of antecedent propositions can be tied to textual order within the framework of ordinary Horn clauses by adding extra arguments to the propositions. This has been done in the theory of *definite clause grammars* (Colmerauer, 1978; Pereira and Warren, 1980).

The above TRADINGVOYAGE rule can be interpreted in either of two ways. If we are given the proposition

```
(TRADINGVOYAGE A RUFOLO WITH GOODS
     IN SHIP TO CYPRUS)
```

in a text, or a summary of a text, we may infer the chain of events

```
(BUY A RUFOLO TH SHIP)
(BUY A RUFOLO TH GOODS)
(LOAD A RUFOLO TH SHIP WITH GOODS)
(SAIL A RUFOLO TO CYPRUS MEANS SHIP)
```

and, if given these events in order in a text, we may infer that Rufolo was performing a TRADINGVOYAGE.

Because of this capability for use in two directions, this rule form can be used both in parsing and generating tasks. A parser can avail itself of these rules to group sets of propositions in a text into units headed by macrostructures (which are the consequents of the rules). These can be further grouped into more generalized macrostructures recursively to yield a story tree. A generator can proceed in the other direction, starting with the top node of a tree and expanding it and its constituents downward recursively (by using the rules that operate on the precondition propositions) to arrive at a tree whose terminals form a coherent text.

2.3 Extended Horn Clauses

After several early experiments with this rule form on a simple text (the Margie story - Rumelhart, 1975) it was discovered that this simple rule form, although capable of handling the computations necessary for the text at hand, did not bring out several inherent attributes that macrostructures generally have in common. For example, in a TRADINGVOYAGE rule, several divisions can be recognized. In order to go on a TRADINGVOYAGE, Rufolo must have the money for a ship and goods and the necessary knowledge about competition to make a profit. Given these, Rufolo will follow a certain sequence of actions; he will obtain a ship and goods, load the goods onto the ship and then sail to Cyprus. The result of this effort will be that Rufolo is in a position to sell or trade his goods (and perhaps make a profit). Therefore, we can break the TRADINGVOYAGE rule into several natural groupings:

```
TRADINGVOYAGE RULE:
HEAD:
(TRADINGVOYAGE A RUFOLO WITH GOODS
     IN SHIP TO CYPRUS)
PRE:
(POSSESS A RUFOLO TH WEALTH)
EXP:
(MAKE A RUFOLO TH
      (CALCULATIONS MOD MERCHANTS
       TYPE USUAL))
(BUY A RUFOLO A SHIP)
(BUY A RUFOLO TH GOODS)
(LOAD A RUFOLO TH SHIP WITH GOODS)
(SAIL A RUFOLO TH SHIP TO CYPRUS)
POST:
(MAKE A RUFOLO TH PROFIT)
```

Structurally, this rule form will be referred to as an "extended" Horn clause (EHC). The first part of the rule is the HEAD of the rule, and represents the macrostructure pattern. The second part is the PREcondition for the rule. The propositions in the precondition are the conditions which must be true, or can be made true, before Rufolo can embark on an episode of TRADINGVOYAGE. The third part is the EXPansion of the rule. If Rufolo goes on a TRADINGVOYAGE, then these are the (probable) actions he will take in doing so. The final part of the rule is the POSTcondition of the rule, which consists of the propositions that will become true upon the successful completion (instantiation) of the TRADINGVOYAGE rule.

The resulting rule form is related conceptually and historically to the notion of a script as developed by Schank and Abelson (1977) (cf. Norman and Rumelhart, 1975). The precondition sets the stage for the invocation of a rule. It describes the setting and the roles of the characters involved in the rule. The expansion consists of the actions normally taken during the invocation of the rule. The postcondition is the result of these actions. When used in a script-like role, a rule is activated when its precondition has been satisfied, and its expansion can then be sequentially instantiated.

A rule can also be used as a plan. A plan is a data structure that suggests actions to be taken in pursuit of some goal. This corresponds to activating a rule according to its postcondition, i.e. employing a rule because its postcondition contains the desired effect. If one has the goal "make money" one might wish to employ the TRADINGVOYAGE rule to achieve it.

This extension of the Horn clause structure serves two purposes. First, by separating the propositions subsumed under a macrostructure into three parts, the EHC rule form renders it unnecessary to label the roles that the individual propositions play in the macrostructure. A macrostructure will usually have preconditions, expansion(s), and a postcondition set, which would have to be labeled (probably functionally) in a simple Horn clause system. Secondly, it serves as a means of separating those propositions which must be true before a rule can be invoked (preconditions) from those whose success or failure follows the invocation of the rule.

A rule may have multiple preconditions if there are several sets of circumstances under which the rule can be invoked. Thus a rule for watching a drive-in movie could have the form

```
WATCH_DRIVE-IN_MOVIE RULE:
HEAD:
(WATCH A PERSON TH DRIVE-IN_MOVIE)
PRE:
(OR ((PERSON IN CAR) (CAR IN DRIVE-IN))
    ((SEE A PERSON TH SCREEN)
    (CAN A PERSON TH
     (READ A PERSON TH LIPS))))
```

A rule may also have several expansions attached to it, one for each potential instantiation of the rule. Thus if a couple want a child they could employ a rule:

```
POSSESS RULE:
HEAD:
(POSSESS A COUPLE TH CHILD)
EXP:
(OR (CONCEIVE A COUPLE TH CHILD)
    (ADOPT A COUPLE TH CHILD)
    (STEAL A COUPLE TH CHILD)
    (BUY A COUPLE TH CHILD))
```

where each of the propositions, CONCEIVE, ADOPT, STEAL, BUY expands by a complex rule of the same form as the POSSESS rule.

A rule can have but a single postcondition, composed of simple propositions, since by definition the postcondition is the set of propositions that are true if the rule succeeds. If a postcondition could contain an expandable proposition, then that proposition could fail independently of the rule for which it is part of the postcondition - since it could have its own preconditions - thus contradicting the definition of the postcondition.

2.4 Indexed Rule Network of Extended Horn Clauses

The rules are stored in memory in a semantic network. However, unlike the usual semantic networks for case predicates, where the individual nodes (tokens) are connected by case relations (Simmons and Chester, 1977), the semantic links in the EHC rule network are based on the story tree structure.

Each rule node (or instantiation of one) in the network may have an arc for each part of an EHC rule: precondition, expansion, and postcondition. All the case relations in the head of the proposition are kept in a single list attached to the node as arc-value pairs. Negation is marked by embedding the node in a (NOT OF ...) proposition form. For example, if a point is reached in a narrative where John is expected to kiss Mary but he does not, then the story tree would contain at that point a node that would print as (NOT OF (KISS A JOHN TH MARY)).

Each node in the database represents either a class object or an instance of a class object. Every class object points to all of the tokens subsumed by it in the network. For example, the class object representing DOG would have pointers to each of its tokens, DOG1, DOG2, DOG3, etc., which represent individual objects of the class DOG.

The database retrieval functions utilize a kind of "fuzzy" partial matching to retrieve potential rules to be applied to a proposition. Partial matching allows the rule-writer to combine rules that only differ in one or two minor arc names, but which all share the same major arc names; only one rule need be written, specifying the major case relations and ignoring the minor ones (which can be checked for in the preconditions of the rules). Partial matching allows the generator to bring more operators to bear at a given point in the story construction. However, the parser pays the price for this flexibility by having to examine more alternatives at each point in its parsing where rules apply.

The function which queries the database for the existence of facts (instantiated propositions), uses a "complete" pattern-matching algorithm, since "John ate cake last night" is not deemed a sufficient answer to the question "Who ate all the cake last night at Mary's place?".

3. Story Analysis Using Extended Horn Clauses

This section describes the use of the Extended Horn Clause rule form and illustrates the process of rule-writing using several paragraphs from the Rufolo story as an example. The notion of rule failure is also presented as an integral feature of parsing and generating narrative texts.

3.1 Writing Rules Using Extended Horn Clauses

The EHC rule form divides the propositions subsumed by a macrostructure into three categories: the preconditions, the expansions, and the postconditions. The preconditions define the applicability of a particular rule. When a precondition has been satisfied, then a rule's expansions and postcondition correspond roughly to the traditional add/delete sets of problem-solving systems. The add set consists of the ordered list of actions to be taken as a result of the rule's invocation plus the postcondition states of the actions. Members of the delete set are propositions of the form (NOT OF NODE) appearing in the expansion and postcondition of a rule.

The question then becomes one of mapping a text into rules of this structure. To illustrate this process consider the following streamlined version of two paragraphs from the Rufolo story:

> Rufolo made the usual calculations that merchants make. He purchased a ship and loaded it with a cargo of goods that he paid for out of his own pocket. He sailed to Cyprus. When he got there he discovered other ships docked carrying the same goods as he had. Therefore he had to sell his goods at bargain prices. He was thus brought to the verge of ruin.
>
> He found a buyer for his ship. He bought a light pirate vessel. He fitted it out with the equipment best suited for his purpose. He then applied himself to the systematic looting of other people's property, especially that of the Turks. After a year he seized and raided many Turkish ships and he had doubled his fortune.

At the top level of the story we have two complex actions taking place: a trading voyage and a pirate voyage. A TRADINGVOYAGE requires a merchant, a ship, some trade goods and a destination on a seacoast (for example, an island). So we embody this in the rule

```
TRADINGVOYAGE RULE:
HEAD:
(TRADINGVOYAGE** A X TH Y TO Z)
PRE:
(X ISA MERCHANT)
(Z ISA ISLAND)
(Y ISA GOODS)
(W ISA SHIP)
```

Given that these conditions are true, then Rufolo plans his strategy, obtains both ship and goods and sails to Cyprus with them (the function of the asterisks is notational only and is used to distinguish those predicates that have rule expansions from those that do not):

```
TRADINGVOYAGE RULE:
HEAD:
(TRADINGVOYAGE** A X TH Y TO Z)
PRE:
(POSSESS* A X TH U)
EXP:
(MAKE* A X TH
     (CALCULATIONS MOD MERCHANT
      NBR PL TYPE USUAL))
(PURCHASE** A X R1 W R2 Y)
(LOAD* A X TH W WITH Y)
(SAIL* A X TH W TO Z)
(TRADE** A X TH Y FOR
     (PROFIT MODAL EXPEC))
POST:
(NOT OF (WANT* A X TH
          (DOUBLE* A X TH U)))
```

```
PURCHASE RULE:
HEAD:
(PURCHASE** A X R1 Y R2 Z)
PRE:
(POSSESS* A X TH (W ISA WEALTH))
EXP:
(POSSESS** A X TH Y)(POSSESS** A X TH Z)
POST:
(NOT OF (POSSESS* A X TH W))
```

```
POSSESS RULE:
HEAD:
(POSSESS** A X TH Y)
PRE:
(POSSESS* A X TH WEALTH)
EXP:
(BUY* A X TH Y)
```

One might consider doing without the PURCHASE rule. Its precondition could have been pushed into the precondition for the TRADINGVOYAGE rule and its expansion inserted into TRADINGVOYAGE at the same point where the PURCHASE is now. But this rule-splitting is not just a stylistic choice; there are several advantages to this strategy. First, keeping rule size small makes the rule easier to read and write. Experience with writing rules showed that an average rule size of three to five propositions proved to be most efficient in this respect. Second, smaller rule size cuts down on the number of permutations of a rule that need to be recognized by the parser, due to missing propositions or the transposition of propositions that describe more or less contemporaneous events. Finally, smaller rules make for more conciseness in summaries generated from the rules.

The postcondition of the PURCHASE rule is that the wealth used to buy the ship and goods is no longer in the possession of its original owner. This would have been inserted into the postcondition of TRADINGVOYAGE if the PURCHASE rule had been incorporated into it directly.

The difference between the POSSESS* and POSSESS** propositions in the PURCHASE rule is illuminating. The POSSESS* in the precondition does not have a rule attached to it, so it cannot be expanded. If a merchant does not already possess wealth when he reaches this point, he cannot go about obtaining it through rule expansion (although the generator can fudge and assert that he does have wealth at this point to move the story generation along). If the precondition of the PURCHASE rule had contained a POSSESS** instead, then the merchant would be able to apply the POSSESS rule above to acquire wealth if he did not have any. The convention, as illustrated above, is for the expandable propositions to be suffixed with two asterisks, and for terminal elements to be suffixed with a single asterisk. This feature is only a notational device.

The TRADE rule describes a trading scenario under conditions of heavy competition:

```
TRADE RULE:
HEAD:
(TRADE** A X TH Y FOR Z)
PRE:
(Z ISA PRICE MOD GOOD)
EXP:
(ADVERSE-TRADE** A X TH Y)
(SELL* A X TH Y FOR Z)
(NOT OF (POSSESS** A X TH Y))
POST:
(MAKE* A X TH (PROFIT MOD GREAT))
```

```
ADVERSE-TRADE RULE:
HEAD:
(ADVERSE-TRADE** A X TH Y)
PRE:
(X ISA MERCHANT)
EXP:
(DISCOVER* A X TH
 (SHIP NBR MANY MOD DOCKED POSSBY
  (MERCHANT NBR SOME MOD OTHER)))
(CARRY* INSTR SHIP TH (GOODS SAMEAS Y))
```

It should be noted that the postcondition of the TRADE rule, the MAKE**, will fail because Rufolo does not SELL* his goods for a profit. Failure in a rule is covered in the next section, and will not be further discussed here.

Because of his failure at trading, Rufolo attempts another scheme to double his wealth (which is his original goal in the story); he turns pirate.

```
PIRATEVOYAGE RULE:
HEAD:
(PIRATEVOYAGE** A X TH Y MEANS Z)
PRE:
(Z ISA SHIP TYPE PIRATE MOD LIGHT)
(POSSESS** A X TH Z)
(Y ISA WEALTH)
(W ISA WEALTH VAL (TWICE* R1 W R2 Y))
(U ISA NATIONALITY)
EXP:
(FITOUT* A X TH Z WITH
 (EQUIPMENT MOD BESTSUITED))
(SEIZE* A X TH
 (SHIP NBR MANY POSSBY U DURATION YEAR))
POST:
(NOT OF (WANT* A X TH
          (DOUBLE* A X TH Y)))
(DOUBLE* A X TH Y)
(POSSESS* A X TH W)
```

The first thing that Rufolo must do is to come to possess a pirate ship, which he does by selling his merchant vessel and using the funds to buy a light vessel:

```
POSSESS RULE:
HEAD:
(POSSESS** A X TH Y)
PRE:
(NOT OF (POSSESS* A X TH WEALTH))
(POSSESS** A X TH Z)
EXP:
(SELL* A X TH Z) (BUY* A X TH Y)
POST:
(NOT OF (POSSESS* A X TH Z))
```

Note that the two POSSESS** rules above would be combined into a single rule in the database.

Rufolo outfits his newly purchased ship as a pirate vessel and then over the course of a year uses it to seize the ships of many Turks, until at the end he has doubled his wealth. The postcondition of this activity is that he no longer wishes to double the amount of his (old) wealth and that he now possesses twice as much wealth as before.

These rules do not thoroughly map the exact meaning of the two paragraphs above, but they do give a flavor for the process of rule-writing, and linguistic precision is easy to ensure by use of additional rules to pin down precisely every nuance in the text, particularly with respect to selling and buying and change of ownership.

Rule-writing using EHCs follows standard top-down programming practice. At the top level are the rules that describe the narrative structure of the text. For the Rufolo story, we have:

```
STORY RULE:
HEAD:
(RUFOLO-STORY** A X)
EXP:
(SETTING A X) (EPISODE)
POST:
(INTEREST* A X TH COMMERCE
     MODAL NOLONGER)
(LIVE* A X MANNER SPLENDOR DURING
          (REMAINDER OF (DAYS OF X)))

EPISODE RULE:
HEAD:
(EPISODE)
EXP:
(OR ((INTERLUDE)(EPISODE))
    ((COMPLICATION)(EPISODE))
    ((COMPLICATION)(RESOLUTION)))
```

The Rufolo story is basically a SETTING followed by an EPISODE, where EPISODE is defined recursively as either an INTERLUDE, which leads into an episode followed by a new EPISODE, a COMPLICATION followed by a new EPISODE, or a COMPLICATION and a RESOLUTION if the COMPLICATION does not cause any new problems to arise (due to rule failure - see next section).

The SETTING rule

```
SETTING RULE:
HEAD:
(SETTING A X)
PRE:
(LIVE* A X LOC Y DURING Z)
(W ISA WEALTH VAL CERTAINAMOUNT)
(POSSESS* A X TH W)
(NOT OF (SATISFY* A X WITH W))
POST:
(WANT* A X TH (DOUBLE* A X TH W))
```

will create a character and his environment - the LIVE* and POSSESS* - and give the character a goal - the WANT*.

Experience with simple narrative text leads to the belief that it is relatively easy to learn to program using EHC rules, and that their expressive power is limited only by the ingenuity of the rule-writer. Experiments with other text forms - encyclopedia articles, magazine articles, dialogues, etc. - need to be performed before a final judgement can be made concerning the fitness of the EHC as a general form for processing.

3.2 Success and Failure in EHC Rules

The idea of rule success or failure in the EHC rule form is tied to the domain the rules attempt to model - real life motivated behavior (actions) where things can, and do, go wrong, and actors fail to achieve their goals. In real life, Rufolo's decision to go on a TRADINGVOYAGE does not guarantee that the voyage will be successful, or even begun. For a given rule, one of three conditions may arise. First the rule's preconditions may fail to be true and the rule cannot be applied. In the TRADINGVOYAGE rule, if Rufolo does not have the money necessary for trading in Cyprus or does not possess the knowledge to successfully compete, then the TRADINGVOYAGE cannot be completed despite Rufolo's best intentions. A rule is not invoked until one of its precondition sets has been successfully processed (i.e. each precondition in the set has been successfully fuzzy-matched).

Once this condition is satisfied then the rule can be invoked and Rufolo can start his TRADINGVOYAGE. At each point of expanding the macrostructure, i.e as Rufolo performs each step in the expansion of the rule, he is subject to success or failure. He may succeed in buying the goods, or he may fail, and the same is true for buying the ship and the loading of the ship with goods, etc. If he does manage to perform all the actions in an expansion, then the rule is said to succeed, and its postcondition propositions can be asserted as being true. In the TRADINGVOYAGE rule, an assertion that Rufolo makes a profit would be instantiated.

But if an expansion fails, a different logic applies. If Rufolo has loaded the ship with his goods, but the ship burns in the harbor, then the last expansion proposition, sailing to Cyprus, cannot be performed. The rule is said to fail, and the postcondition propositions are negated. In the TRADINGVOYAGE rule, an assertion that Rufolo does not make a profit would be instantiated because no other expansion options remain.

Rule failure is an important concept with respect to narratives. Many narratives consist of a series of episodes in pursuit of a goal; often each episode, except the last, represents an instance of a failed rule. If a rule prior to the last succeeded, then the goal would have been achieved and no further episodes would have been forthcoming, (unless a new goal were chosen). The mechanism of rule failure is a natural one for analyzing this type of narrative pattern.

4. Generation and Parsing

In this section a procedure for using macrostructures, embodied in the EHC rule-form, to generate and parse stories, and a program, BUILDTALE/TELLTALE, that implements it are discussed.

4.1 Generation

The paradigm used by TELLTALE for story generation is similar to that used by Meehan (1976) in his TALESPIN program, which wrote "metanovels" concerning the interrelationships, both physical and mental, of the motives and actions of anthropomorphized animals possessing simple behavior patterns. Meehan described a story as an exposition of events that occur when a motivated creature attempts to achieve some goal or fulfill some purpose, such as satisfying a need or desire, solving a problem, doing a job, etc. TALESPIN was basically a problem-solver that operated on a database consisting of living entities, inanimate objects, and a set of assertions (rules) typifying the relationships among them, and describing operations for affecting those relationships. The rules were organized into plans based on conceptual relatedness, which aided in retrieval of specific rules.

Meehan emphasized the use of plans as a mechanism for generating stories. Plan activation was a process of assigning a goal to a character and then calling up any relevant plans that were indexed by this goal.

In the EHC rule form the precondition governs the ability of a rule to be invoked. If a rule is considered to be a plan in the Meehan sense above, then instead of indexing the rule by its postcondition propositions, it can be indexed by its precondition if the precondition contains information relating to motivation. For example, the rule,

```
MARRY RULE:
HEAD:
(ASK-MARRY** A X TH Y)
PRE:
(X ISA MAN)
(Y ISA WOMAN)
(WANT* A X TH (MARRY* A Y TH X))
EXP:
(GO* A X TO Y)
(ASK* A X TH Y IF (MARRY* A Y TH X))
(ACCEPT* A Y TH X)
POST:
(MARRY* A Y TH X)
```

is a rule that can be activated if some X, who must be a man, WANTs Y, who must be a woman, to marry X. Each rule could contain propositions in its preconditions that restrict the set of objects that can instantiate the variables of the rule.

The last proposition in the precondition, the WANT*, is a goal statement; it states that the MARRY rule can be invoked when one person wants to marry another. In the process of generating a story, if a point is reached where a man develops the goal of wanting to marry a woman, the MARRY rule can be

invoked to attempt to satisfy that want. Of course, if the ASK* or ACCEPT* fails, then the postcondition becomes (NOT OF (MARRY* ...)) instead.

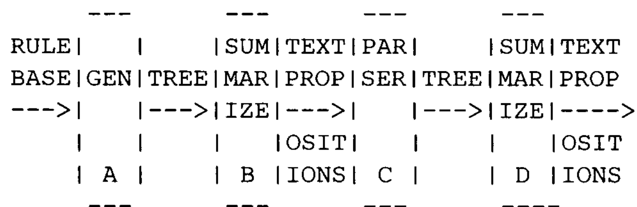
The main advantage of inserting the WANT* into the preconditions is that the generator need not, as a result, contain special logic to examine the postconditions of a rule to ascertain which rules to apply at a point in the story. The normal rule activation mechanism (a rule can only be applied if one of its precondition sets succeeds) will automatically weed out unwanted rules because of the information relating to motivation.

TELLTALE generates stories (sequences of propositions) based on such rules contained in its database, either under user control or directed by the program itself. The database is a semantic network of nodes created by building a network of the case-predicate rules supplied by the user. The input to TELLTALE is the root of a story tree, which informs TELLTALE as to the type of story which is to be generated (fairy tale, etc.). The output from TELLTALE is an instantiated story tree whose terminals are the propositions of the text. The program SUMMARIZE computes summaries from the story tree. An annotated example rule-base can be found in Appendix A for generating a set of fairy tales. A sequence of propositions for a story generated from that rule-base is shown in Appendix B. Two summaries of the story are shown in Appendix C.

Starting with the root, TELLTALE executes until no nodes remain to be expanded, because all have been reduced to terminal propositions or some depth limit has been reached. A traversal of the terminals of the resulting story will yield the proposition text of the story; any higher level traversal will yield a summary of the story.

4.2 Parsing

This section describes a procedure, BUILDTALE, for using macrostructures to parse stories. The goal in writing the generator-parser system has been to create a program capable of understanding those texts that it can generate from a rule-base. Diagrammatically



The tree that results from the story generator should yield the same bottom-level terminal propositions as the tree that results from the parsing of the terminal propositions of the story, i.e. the output from part B in the diagram above should be the same as the output from part D. The story trees might or might not be identical, depending on whether the story grammar is ambiguous.

The philosophy of the understander can be summarized by recalling a concluding statement by Rumelhart (1975):

> "It is my suspicion that any automatic 'story parser' would require ... 'top-down' global structures ..., but would to a large degree discover them, in any given story, by the procedures developed by Schank (1975)."

The procedures developed by Schank emphasize the "bottom-up" approach to story parsing. Starting with the raw input data, emphasis is placed on integrating each sentence into the developing narrative structure. The structures used are scripts (and plans), and parsing integrates propositions into scripts. The scripts form an implicit episodic structure, but the higher order relationships among the scripts are not generally made specific (i.e. what structures form episodes, setting information, complications, resolutions, etc).

BUILDTALE is a program that combines the top-down and bottom-up approaches. As the parse progresses, BUILDTALE attempts to integrate the terminal propositions into the context of a story tree. It manipulates the terminals bottom-up, integrating them into macrostructures, while it is building story structure nodes (STORY, SETTING, EPISODE, etc.) top-down. If BUILDTALE successfully parses the text, these two processes will converge to instantiate a complete story tree.

Starting with the root of a story tree and an unmarked string of text propositions, BUILDTALE executes, by an algorithm described in Section 4.4, until one of three conditions occurs:

1) If the procedure exhausts the proposition list before the STORY node has been built, then it fails.
2) If the procedure builds an instance of STORY but there still remain unmarked propositions, then it fails.
3) If an instance of STORY is built and the text stream is empty, the the procedure returns the instantiated root.

A terminal-level traversal of the resulting tree will yield the original input text proposition stream; higher level traversals will yield summaries of the text.

4.3 The Relationship Between Generating and Parsing

There are several differences between the BUILDTALE and TELLTALE procedures. First, whereas the parser is restricted to propositions from the input

text for its terminals, the generator is free to build a terminal any time one can be generated. Second, the generator is free at each step to choose any rule that will match a proposition it is trying to expand, and also to use any of a rule's preconditions or expansions in any order. The parser must be careful in its choice of rules and in how it examines the preconditions and expansions in a rule, always examining them in order of decreasing length, to ensure that it does not build a story tree and find left-over unmarked text propositions when done.

The generator builds a proposition by first instantiating its precondition, expansion, and postcondition, and then attaching them to the instantiated head. The generator knows at all times the father of the proposition it is instantiating; in fact it knows every ancestor of that proposition between it and the root, since it operates strictly top-down.

The parser operates primarily as a left-corner bottom-up procedure, with overall direction supplied by some top-down processing. When the parser builds a structure, it cannot be sure at that time that the structure is indeed the correct one to be integrated into the tree at that point, i.e. it does not yet know the correct path to the root. The parser must, therefore, save the context information of its build decisions, so that they can be undone (or at least ignored) if they are later found to be in error. In fact, the final structure of the tree can only be assigned after all the textual propositions have been analyzed. This is in agreement with the statement of van Dijk (1975, pg. 11):

> "Strictly speaking, a definite hierarchical structure may be assigned to a discourse sequence of propositions only after processing of the last propositions. For long discourses this would mean that all other propositions are kept in some memory store."

Structures are built as the parse progresses, and some might be discarded or rearranged. Their final position in the tree cannot be determined until all the propositions have been examined.

Some previous parsers solved this problem by resorting to higher-level languages like CONNIVER and PLANNER, paying the price in higher computational costs. A conscious effort was made with this project to avoid the expense of resorting to a higher-level language by having LISP perform the necessary bookkeeping to handle the backtracking involved in undoing an incorrect choice (build). In BUILDTALE, the bookkeeping is accomplished by pushing context information onto the LISP control stack. Usually, when a build is performed , instead of returning (popping the LISP stack), a further descent is made in order to integrate the next proposition. If a build is later found to be in error, then a FAIL function automatically causes LISP to back up in its stack to the point where the build was made and undo it, since all the information that was around when the first decision was made to build is still present on the stack.

These differences should not obscure the very real similarities between the two processes. TELLTALE and BUILDTALE use the same functions to analyze the preconditions, expansions and the postcondition of a rule. In fact, the "basic" control structure of TELLTALE is a special case of the control structure of BUILDTALE. The difference between the two occurs at build time. In BUILDTALE, when a node in the tree is built, a check is made to see if this node matches the root of the derivation tree being built. This may not be the case since the node may be many levels lower in the tree than the root in question, and these levels will need to be built before the derivation tree is complete. Of course, if the node should match the root, then it is returned.

TELLTALE, on the other hand, never descends more than a single level in the tree at a time. When a build is performed, it will always derive from the derivation tree being processed. The node and the root always match, and the node is returned. At build time, when BUILDTALE decides whether to call itself recursively (to add the next higher level in the tree) or to pop the stack (returning the derivation tree root), TELLTALE will always pop the stack.

Generation and parsing use the same grammar with different terminating conditions to define the control paths through the grammar. They resemble each other in computing as output a derivation tree whose terminals are the propositions of the text. This fact was borne out during the implementation of the system. TELLTALE was coded first, and the eventual TELLTALE/BUILDTALE control structure for processing preconditions, expansions, and postconditions was debugged and tested by generating many story trees. BUILDTALE grew out of TELLTALE by adding the build-time recursion/backup mechanism to the control structure.

The symmetric relationship between generation and parsing with respect to the computation of derivation trees is one significant feature of the EHC rule system.

4.4 The Basic Control Structure

BUILDTALE and TELLTALE are essentially large separate PROG-defined LISP functions that effect different initialization conditions, and then execute the same set of underlying functions. Below is a description of the basic control structure shared by TELLTALE and BUILDTALE:

I: For each proposition being examined:
- 1) If it exists instantiated in the data base, then return the instantiation, whether it is found negated or not;
- 2) if there are no rules for the node, then it can be asserted;
- 3) otherwise, for each rule until one succeeds:
  - a) for each precondition in the rule until one succeeds, do I for each proposition in the precondition;
  - b) if no precondition succeeds, then fail;
  - c) otherwise, for each expansion until one succeeds, do I for each proposition in the expansion;
  - d) if an expansion succeeds, then for each proposition in the postcondition, do I and return the instantiated node (rule success);
  - e) otherwise, for each proposition in the postcondition, do I for the negation of the proposition and generate a (rule) failure.

"Do I" means "follow the procedure labelled 'I'" - *which is the entire basic control structure outlined above.* Since postcondition propositions do not have rule expansions, they never perform step 3 above. Also, rule failure is not marked by negation (another mechanism is used) so, if a negated node is encountered, it will never fall through to step 3. Finally, it should be noticed that, although it is possible for the algorithm to generate erroneous propositions in step 2 if exploring an incorrect path during parsing (BUILDTALE), these propositions will be undone during backup when the path has been recognized as incorrect.

5. Extracting Summaries From Story Trees

One of the principal reasons for choosing the Kintsch and van Dijk macrostructure theory was the resulting property of summarizability; the ability to produce coherent summaries is one mark of intelligence in a parser. The summarizer produces various strings of propositions from the story tree which form summaries of a text. One such string is composed of the terminals and represents the complete story. Any sequence of propositions output by the summarizer is a well-formed input to the parser. The system is thus able to parse all proposition sequences it can generate.

Since the summary feature is inherent in the trees as they are given to the summarizer, a simple level-traversal algorithm would have been sufficient to generate useful output. However, this would have resulted in needless redundancy of propositions (since some are checked repeatedly by the preconditions of the rules and have pointers to them at many places in the tree). Therefore, the summarizer remembers what it has already output in the summary, so as never to repeat itself.

Another area of repetition concerns attributes for objects in the story. To avoid repeating an object's attributes, the summarizer keeps a list of objects that have already appeared in at least one proposition, and whenever it encounters in a new proposition an object not on this list, it outputs it with all of its properties and then places it on the list of expanded objects. Since no time-markers are put on an object's properties, they are all printed out at once, even if some of those properties are not attached to the object until much later in the story; this reveals a weakness in the procedure that can be corrected by the introduction of time-markers for objects (actions already possess time-markers).

One attribute of story trees is that, at their higher nodes, one can read off the syntactic structure of the story. For example suppose a story consists of a setting followed by three episodes. As a summary, "setting plus three episodes" is usually not very interesting; therefore the summarizer has the ability to recognize and descend below these story structure nodes in the final summary. These nodes are treated like all other nodes to the tree building procedures, but the summarizer descends below the nodes to print their descendants, no matter what level summary is being computed.

There is also the question of summarizing the macrostructures (rules). By definition, these nodes are expandable, i.e., they have a rule for expanding them in the rule-base. Macrostructures are not marked with a NOT if they fail; only simple propositions - terminals - are. However, whether a script achieves its goal or not is vital information to be included in any reasonable summary produced from the tree. Therefore, when summarizing a macrostructure, the summarizer outputs both the head (the macrostructure pattern) and its postcondition (if the script fails to achieve its goal, the postcondition will have been negated).

Finally, the summarizer outputs only those propositions it recognizes as non-stative descriptions; it never outputs state descriptions. The reason for this is that a stative always describes the attribute(s) of some object, and can therefore be output the first time that that object appears in an active proposition.

The summarizer is an algorithm that, given a story tree and a level indicator, scans the nodes of the tree at that level, and applies the following rules to each node:

1) If the node is a story structure node, then summarize its sons.
2) If the node has already been processed, then skip it.
3) If the node is marked as a script, return its head followed by its postcondition propositions.
4) If the node is a stative, then skip it.

For each case value in a proposition to be output (for example, RUFOLO1, DOUBLE*1, and WEALTH1 in (WANT*1 A RUFOLO1 TH (DOUBLE*1 A RUFOLO1 TH WEALTH1))), the following rule is applied:

5) If the case value has not appeared in a previously accepted proposition, print it and all its attributes; otherwise, just print the case value itself.

For example, if RUFOLO1 has been mentioned in a proposition prior to WANT*1 in a summary, then only RUFOLO1 will be present in the WANT*1 proposition, and not the fact that RUFOLO1 is a MERCHANT1 from RAVELLO1, etc.

An initial version of an English language generator was written that applies a set of rules to the output of the summarizer to produce well-formed English texts (Hare and Correira, 1978). This generator uses rule forms and the summarizer output to solve some of the problems involved in: reasonable paragraphing and sentence connectivity, elision of noun groups and verbs, and pronominalization of nouns. The decisions are based, in part, on the structure of the story tree and the positions of individual propositions in that tree.

6. Discussion and Conclusions

The task of text processing requires solutions to several important problems. The computational theory for macrostructures using Extended Horn Clauses was designed with the following goals in mind. A computational model should have some degree of psychological validity, both to provide a humanly useful representation of textual content and organization and to ensure that the task of rule-writing is as natural as possible for the human grammar producer. It should be conceptually simple, in both design and implementation, taking advantage of similarities between generation and parsing, and it should offer a rigorous data structure that is uniform and avoids the growth of ad hoc large-scale structures.

The computational macrostructures realized by the Extended Horn Clause notation succeed in many ways in satisfying these goals. They appear to resemble the structures humans build mentally when reading narrative texts. The story tree is a logical way to organize these macrostructures, with the terminals of a particular story tree comprising the actual textual propositions, and the interior nodes containing the instantiated heads of rules (corresponding to macrostructures). The story tree has the summary property: if the tree is truncated at any level, then a "meaningful" (coherent) summary of the original text can be read off directly. The generality of the macrostructure propositions increases as one nears the level of the root (going from the level of specific actions to the rules that contain them, to the story categories that contain these rules), which can be considered as the title for the text at its terminals.

The concept of rule failure takes the EHC out of the strictly logical system of the normal (Kowalski-type) Horn clause logic, since failure in a normal logic system means something different from failure here. In narratives, failure needs to be recorded, since it is one source of "interest" in the resulting story; striving, failing, and striving again is a common occurrence in simple narratives. These failure points, and their consequences, have to be recorded in the story tree (whereas, in normal logic systems, failure points are invisible to the final result) and, furthermore, they restrict the choice of paths that can reasonably be expected to emanate from the point of failure. The failure mechanism is tailored for narratives involving entities exhibiting motivated behavior. Other text forms, such as technical or encyclopedia articles, would probably not require the failure mechanism.

The underlying approach in BUILDTALE/TELLTALE is that of a problem-solver, as was also true of Meehan's story-writer. A rule-base, organized as a hierarchy of story trees, is used to generate a particular, instantiated, story tree by an inference procedure augmented via the rule failure mechanism detailed in Section 3.2. Each instantiated tree is treated as a context, consisting of the events, objects, and their relationships, relating to a particular story. The facts and rules in the rule-base serve as a model for the possible states in the microworld formed by that story tree. These states are categorized using linguistic entities, such as SETTING, EPISODE, COMPLICATION, and RESOLUTION.

The problem-solving approach, coupled with the story grammar concept, is a natural one for processing most forms of narratives. Analogous systems of rules could be used for processing other large text forms, although the categories involved would be different.

Notes on the Appendices

The rules are all written in a case predicate notation (Fillmore, 1968). The general form for such a predicate is

```
(HEAD ARC1 VALUE1 ... ARC[n] VALUE[n])
```

The HEAD of a case predicate is either a verb or a noun form; because no formal lexicon was maintained for the TELLTALE/BUILDTALE program, verb forms were marked with an asterisk and objects were left unmarked. The ARCs are standard case relations,

such as Agent, THeme, LOCation, INSTance, etc., although no attempt was made to be strictly correct linguistically with their every use, and a few relations were created for the sake of convenience. When a good arc name did not suggest itself, then an arbitrary arc name - R1, R2, etc. - was used instead. The VALUE can be either a verb form, an object, or another case predicate.

The case predicates used in the program were written to enhance readability. For example, in the fairytale story (Appendix B), the case predicate

```
(WANT*1 TO POSSESS1 A GEORGE1 TH MARY1)
```

can be rendered into English as "George wants to possess Mary". The sequence

```
(GO*3 A GEORGE1 TO IRVING1)
(SLAY*1 A GEORGE1 TH IRVING1)
(RESCUE*1 A GEORGE1 TH MARY1)
```

can be rendered as "George goes to Irving. George slays Irving. George rescues Mary.".

Appendix A - Rule-base for Fairytale

```
FAIRYTALE RULE:
HEAD:
(FAIRYTALE*)
EXP:
(FAIRYSTORY** A X TH Y)

FAIRYSTORY RULE:
HEAD:
(FAIRYSTORY** A X TH Y)
EXP:
(SETTING A X)
(EPISODE A X TH Y)
POST:
(LIVE* A X TH Y MANNER
     HAPPILY_EVER_AFTER)

SETTING RULE:
HEAD:
(SETTING A X)
PRE:
(LIVE* A X LOC Y DURING Z)

LIVE RULE:
HEAD:
(LIVE* A X LOC Y DURING Z)
PRE:
(CHAR INST X) (Y ISA PLACE) (Z ISA TIME)

CHAR RULE:
HEAD:
(CHAR INST X)
PRE:
(OR (X ISA KNIGHT SEX MALE PERSON T)
    (X ISA PRINCE SEX MALE PERSON T))
EXP:
(OR (X MOD BRAVE) (X MOD HANDSOME))

EPISODE RULE:
HEAD:
(EPISODE A X TH Y)
EXP:
(MOTIVE A X TH Y) (ACTION A X TH Y)

MOTIVE RULE:
HEAD:
(MOTIVE A X TH Y)
PRE:
(DESIRE* A X TH Y)
EXP:
(WANT* TO POSSESS A X TH Y)

DESIRE RULE:
HEAD:
(DESIRE A X TH Y)
PRE:
(CHAR INST X)
EXP:
(OR (Y ISA PRINCESS SEX FEMALE
     PERSON T MOD BEAUTIFUL)
    (Y ISA HOLY_OBJECT MOD LOST))

ACTION RULE:
HEAD:
(ACTION A X TH Y)
EXP:
(OR (ASK-MARRY** A X TH Y)
    (RESCUE** A X TH Y FROM Z)
    (QUEST** A X TH Y)
    (PRAY** PART FOR A X TH Y))

ASK RULE:
HEAD:
(ASK-MARRY** A X TH Y)
PRE:
(WANT* TO POSSESS A X TH Y)
(Y ISA PRINCESS)
EXP:
(GO* A X TO Y)
(ASK* A X TH Y IF (MARRY* A Y TH X))
(ACCEPT* A Y TH X)
POST:
(MARRY* A Y TH X)

RESCUE RULE:
HEAD:
(RESCUE** A X TH Y FROM Z)
PRE:
(WANT* TO POSSESS A X TH Y)
(Y ISA PRINCESS)
(THREATEN** A Z TH Y)
EXP:
(GO* A X TO Z)
(SLAY* A X TH Z)
(RESCUE* A X TH Y)
POST:
(MARRY* A Y TH X)
```

```
THREATEN RULE:
HEAD:
(THREATEN** A X TH Y)
PRE:
(X ISA DRAGON ANIMATE T)
(X MOD EVIL)
(Y ISA PRINCESS)
(WANT* TO POSSESS A X TH Y)
EXP:
(CARRY** PART OFF A X TH Y TO Z)

CARRY RULE:
HEAD:
(CARRY** PART OFF A X TH Y TO Z)
PRE:
(X ISA DRAGON)
(Y ISA PRINCESS)
(Z ISA  DEN POBJ T)
EXP:
(GO* A X TO Y)
(CAPTURE* A X TH Y)
(FLY* A X TH Y TO Z)

QUEST RULE:
HEAD:
(QUEST** A X TH Y)
PRE:
(CHAR INST X)
(Y ISA HOLY_OBJECT MOD LOST)
(WANT* TO POSSESS A X TH Y)
EXP:
(GO* A X TO ORACLE)
(REVEAL* A ORACLE TH PLACE OF Y)
(GO* A  X TO PLACE)
(FIND* A X TH Y)
POST:
(POSSESS* A X TH Y)

PRAY RULE:
HEAD:
(PRAY** PART FOR A X TH Y)
PRE:
(CHAR INST X)
(WANT* TO POSSESS A X TH Y)
(Z ISA GOD)  (W ISA CHURCH POBJ T)
EXP:
(OR ((GO* A X TO W)
     (KNEEL* A X PREP (IN TH
      (FRONT PREP (OF TH ALTER))))
     (PRAY* A X PREP (TO TH Z)
      PREP (FOR TH Y))
    ((U ISA PRIEST SEX MALE PERSON T)
     (GO* A X TO U)
     (PAY* A X TH U EXPECT
      (INTERCEDE* A U PREP (WITH TH Z)
       PREP (FOR TH Y)))
     (PRAY* A U TO Z FOR Y)
     (GRANT* A Z TH (PRAYER POSSBY U))))
POST:
(POSSESS* A X TH Y)
```

```
(JOHN ISA PRINCE SEX MALE PERSON T)
(GEORGE ISA KNIGHT SEX MALE PERSON T)
(LANCELOT ISA KNIGHT SEX MALE PERSON T)
(PARSIFAL ISA KNIGHT SEX MALE PERSON T)

(MARY ISA PRINCESS SEX FEMALE PERSON T)
(GUENEVIERE ISA PRINCESS
 SEX FEMALE PERSON T)

(HOLY_GRAIL ISA HOLY_OBJECT POBJ T)
(SACRED_CROSS ISA HOLY_OBJECT POBJ T)

(CAMELOT ISA PLACE)
(MONTSALVAT ISA PLACE)

(ONCE_UPON_A_TIME ISA TIME)

(IRVING ISA DRAGON ANIMATE T)
(CARMEN ISA DRAGON ANIMATE T)
```

Appendix B - Text of Fairytale

```
(FAIRYTALE*1)

((LIVE*2 A (GEORGE1 ISA KNIGHT1 SEX MALE2
  PERSON T MOD BRAVE1) LOC (CAMELOT1 ISA
  PLACE1) DURING (ONCE_UPON_A_TIME1 ISA
  TIME1))
 (DESIRE*2 A GEORGE1 TH (MARY1 ISA
  PRINCESS1 SEX FEMALE1 PERSON T
  MOD BEAUTIFUL1))
 (WANT*1 TO POSSESS1 A GEORGE1 TH MARY1)
 (GO*1 PART TO1 A GEORGE1 TO MARY1))
 (ASK*1 A GEORGE1 TH MARY1 IF
  (MARRY*1 A MARY1 TH GEORGE1))
 (NOT OF (ACCEPT*1 A MARY1 TH GEORGE1))
 (NOT OF (MARRY*2 A MARY1 TH GEORGE1))
 (WANT*2 TO POSSESS2 A
  (IRVING1 ISA DRAGON1 ANIMATE T
   MOD EVIL1) TH MARY1)
 (GO*2 PART TO2 A IRVING1 TO MARY1)
 (CAPTURE*1 A IRVING1 TH MARY1)
 (FLY*1 A IRVING1 TH MARY1 PREP
  (TO TH DEN1))
 (GO*3 PART TO3 A GEORGE1 TO IRVING1)
 (SLAY*1 A GEORGE1 TH IRVING1)
 (RESCUE*1 A GEORGE1 TH MARY1)
 (MARRY*4 A MARY1 TH GEORGE1)
 (LIVE*3 A GEORGE1 TH MARY1
  MANNER HAPPILY_EVER_AFTER1))
```

Appendix C - Summaries of Fairytale

```
(FAIRYTALE*1)

((LIVE*2 A (GEORGE1 ISA KNIGHT1 SEX MALE2
  PERSON T MOD BRAVE1) LOC (CAMELOT1 ISA
  PLACE1) DURING (ONCE_UPON_TIME1 ISA
  TIME1))
 (DESIRE*2 A GEORGE1 TH (MARY1 ISA
  PRINCESS1 SEX FEMALE1 PERSON T
  MOD BEAUTIFUL1))
```

```
(WANT*1 TO POSSESS*1 A GEORGE1 TH MARY1)
(GO*1 PART TO1 A GEORGE1 TO MARY1)
(ASK*1 A GEORGE1 TH MARY1 IF
 (MARRY*1 A MARY1 TH GEORGE1))
(NOT OF (ACCEPT*1 A MARY1 TH GEORGE1))
(NOT OF (MARRY*2 A MARY1 TH GEORGE1))
(WANT*2 TO POSSESS2 A
 (IRVING1 ISA DRAGON1 MOD EVIL1)
 TH MARY1)
(CARRY**2 PART OFF1 A IRVING1
 TH MARY1 TO DEN1)
(GO*3 PART TO3 A GEORGE1 TO IRVING1)
(SLAY*1 A GEORGE1 TH IRVING1)
(RESCUE*1 A GEORGE1 TH MARY1)
(MARRY*4 A MARY1 TH GEORGE1)
(LIVE*3 A GEORGE1 TH MARY1
 MANNER HAPPILY_EVER_AFTER1))

(FAIRYTALE*1)
((LIVE*2 A (GEORGE1 ISA KNIGHT1 SEX MALE2
 PERSON T MOD BRAVE1) LOC (CAMELOT1 ISA
 PLACE1) DURING (ONCE_UPON_A_TIME1 ISA
 TIME1))
(DESIRE*2 A GEORGE1 TH (MARY1 ISA
 PRINCESS1 SEX FEMALE1 PERSON T
 MOD BEAUTIFUL1))
(WANT*1 TO POSSESS1 A GEORGE1 TH MARY1)
(ASK-MARRY**2 A GEORGE1 TH MARY1)
(NOT OF (MARRY*2 A MARY1 TH GEORGE1))
(RESCUE**2 A GEORGE1 TH MARY1
 FROM (IRVING1 ISA DRAGON1 ANIMATE T
 MOD EVIL1))
(MARRY*4 A MARY1 TH GEORGE1)
(LIVE*3 A GEORGE1 TH MARY1
 MANNER HAPPILY_EVER_AFTER1))
```

References

Bobrow, D. G. and Collins, A. 1975. *Representation and Understanding*. New York: Academic Press.

Bobrow, D. G. and Raphael, B. 1974. "New programming languages for AI research". *Computing Surveys*, Vol. 6, No. 3, pp. 155-174.

Bobrow, D. and Winograd, T. 1977. "An overview of KRL, a knowledge representation language." *Cognitive Science*, Vol. 1, No. 1, pp. 3-46.

Charniak, E. 1972. "Toward a model of children's story comprehension." AI-TR-266. Cambridge, Mass.: M.I.T. A.I. Lab.

Charniak, E. and Wilks, Y. 1976. *Computational Semantics*. New York: North-Holland.

Colmerauer, A. 1978. "Metamorphosis grammars" in *Natural Language Communication with Computers*, ed. L. Bolc, New York: Springer Verlag.

Fillmore, C. 1968. "The case for case" in *Universals in Linguistic Theory*, eds. E. Bach and R. Harms, New York: Holt, Rhinehart and Winston.

Grosz, B. 1977. "The representation and use of focus in dialogue understanding." Technical Note No. 151. Menlo Park, California: Stanford Research Institute.

Hare, D. and Correira, A. 1978. "Generating connected natural language from case predicate story trees." unpublished manuscript.

Hendrix, G. G. 1976. "Partitioned networks for modelling natural language semantics." Dissertation. Austin, Texas: Department of Computer Sciences, The University of Texas.

Kintsch, W. and van Dijk, T. 1978. "Recalling and summarizing stories." in *Current Trends in Textlinguistics*, ed. Wolfgang Dressler, de Gruyter.

Kowalski, R. 1979. *Logic for Problem Solving*, New York: Elsevier North-Holland.

Lehnert, W. 1977. "Question answering in a story understanding system." *Cognitive Science*, Vol. 1, pp. 47-73.

Meehan, J. 1976. "The metanovel: writing stories by computer." Dissertation. New Haven, Connecticut: Department of Computer Sciences, Yale University.

Meyer, B. 1975. *The Organization of Prose and its Effects on Recall*. Amsterdam: North-Holland.

Minsky, M. 1975. "A framework for representing knowledge." In *The Psychology of Computer Vision*. P. Winston *ed*. New York: McGraw-Hill.

Norman, D. A. and Rumelhart, D. E. 1975. *Explorations in Cognition*. San Francisco: W. H. Freeman and Company.

Pereira, F. and Warren, D. 1980. "Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks". *Artificial Intelligence 13*, pp. 231-278.

Rumelhart, D. E. 1975. "Notes on a schema for stories." in *Representation and Understanding*, eds. D. Bobrow and A. Collins, New York: Academic Press.

Schank, R. C. 1975. "The structure of episodes in memory." in *Representation and Understanding*, eds. D. Bobrow and A. Collins, New York: Academic Press.

Schank, R. C. 1975b. *Conceptual Information Processing*. New York: North-Holland.

Schank, R. and Abelson, R. 1977. *Scripts, Plans, Goals and Understanding*. New York: Wiley.

Simmons, R. F. 1978. "Rule-based computations on English." in *Pattern-Directed Inference Systems*. Hayes-Roth, R. and Waterman, D. eds. New York: Academic Press.

Simmons, R. F. and Correira, A. 1979. "Rule forms for verse, sentences and story trees." in *Associative Networks - Representation and Use of Knowledge by Computers*, ed. N. Findler, New York: Academic Press.

Simmons, R. F. and Chester, D. 1977. "Inferences in quantified semantic networks". *Proceedings of Fifth IJCAI*, pp. 267-274.

van Dijk, T. A. 1975. "Recalling and summarizing complex discourse." unpublished manuscript. Amsterdam: Department of General Literary Studies, University of Amsterdam.

Wilks, Y. 1975. "A preferential, pattern-matching semantics for natural language understanding". *Artificial Intelligence* 6, pp. 53-74.

Winograd, T. 1972. *Understanding Natural Language*. New York: Academic Press.

Woods, W. A. 1970. "Transition networks grammars for natural language analysis". *Comm. ACM*, Vol. 13, pp. 591-602.

Young, R. 1977. "Text Understanding: a survey." *American Journal of Computational Linguistics*, Vol. 4, No. 4, Microfiche 70.

Alfred Correira is a Systems Analyst for the Computation Center of the University of Texas at Austin. He received the M.A. degree in computer science from the University of Texas in 1979.