A NATURAL LANGUAGE INTERFACE FOR THE QED TEXT EDITOR

by

Ruth Marzi

Submitted to the Department of Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Professor in Charge

Committee Members

For the Department

Date thesis accepted

ABSTRACT

In this study a natural language interface for the QED text editor has been designed and implemented in LISP on the Honeywell 66/60 computer. It is based on W. Woods' theory of augmented transition networks. The program is aimed at simulating the manipulation of a textfile. The file is assumed to contain a document which is subdivided into paragraphs. The input consists of a sequence of natural language sentences, each of them a command. The program transforms these sentences into syntactically correct QED commands. At the end of a session a summary of the commands is given and information is provided about the structure of the changed document.

The program is not capable of handling the full variety of commands usually available in text editors. The fact that program directed interaction with the QED text editor is not possible makes dealing with certain types of commands impossible. Also since LISP was chosen as implementation language, some of its features hinder processing commands in some cases. Lastly, a program that is capable of handling the full variety of commands is too complex in this environment.

TO MY PARENTS

ACKNOWLEDEMENTS

I wish to thank the University of Bonn for giving me a scholarship that enabled me to come to the University of Kansas. I also want to thank the Graduate School of the University of Kansas for extending the scholarship so that I could finish the degree of Master of Science.

I wish to express my gratitude to Dr. Albert Bethke who guided my troughout my study and without whose help this project would not have been possible. I also want to thank Dr. Jerzy Grzymala-Busse and Dr. Adrian Tang for agreeing to serve as committee members.

TABLE OF CONTENTS

1. Overview of natural language processing approaches

There are several ways of processing natural language sentences. One approach to parsing natural language is using augmented transition networks (ATNs)[3]. Transformational grammars which were introduced by Noam Chomsky in 1957 [1] are another approach. Also there is DIAGRAM by Jane J. Robinson for interpreting English dialogue [5]. Furthermore, in the following, the conceptual dependency theory by R. Schank [6,7] and predecessors of ATNs [3] are mentioned.

1.1. Keyword Matching

There are basically two types of parsers for natural languages. The first group of parsers ignores syntax entirely as in STUDENT by Bobrow and ELIZA by Weizenbaum. These systems either allow only a small set of fixed input form, or limit understanding to what can be handled without analysing syntax. For example in ELIZA, if the parser finds the keyword 'mother', the program replies with "Tell me more about your family.", no matter what the rest of the input sentence looks like. The second group takes a simplified subset of English which can be handled by a well-understood grammar like a context free grammar or one of its variations. These systems are not capable of handling the full complexity of natural language though.

1.2. Chomsky's Hierarchy

Chomsky defined three classes of grammars as potential models of natural language [1].

1. regular grammars
2. phrase structure grammars
3. transformational grammars.

Phrase structure grammars are subdivided into context sensitive grammars and context free grammars. A transformational grammar is a grammar for English based on rewrite rules. A sentence is rewritten as a simpler sentence or as several simpler sentences. Since transformational grammars bear no significance for formal languages, in formal language theory grammars are divided in a different way [8].

In the following a grammar is a 4-tuple G=(V,T,P,S)
where

V - the set of variables or non-terminal symbols

T - the set of terminal symbols

P - a production system

S - the start symbol, a symbol of V.

The classes are distinguished by the restrictions on their production system. These classes are:

1. Type 0 or unrestricted grammars
2. Type 1 or context-sensitive grammars

3. Type 2 or context-free grammars
4. Type 3 or regular grammars.

Except for the empty string each class is, in decending numerical order, a proper subset of the class succeeding it.

Type 0 grammars define the largest family of languages. These grammars, which are also called semi Thue grammars, characterize the recursively enumerable sets. Languages produced by type 0 grammars are equivalent to languages recognized by Turing machines. The productions of a type 0 grammar are of the form

A -> B

where A and B are arbitrary strings of terminal and non-terminal symbols and A is not the empty string.

Type 1 grammars define phrase-structure languages with restrictions. The productions are length preserving, i.e. in a production

A - > B

(A, B as above) the string B has to be at least as long as the string A. So these grammars cannot produce the empty string. Context-sensitive languages, which are produced by type 1 grammars, are recognized by linear bounded automata (LBA) except for the empty string. A LBA is a Turing Machine restricted to the portion of the input tape containing the input and two squares at either end for special symbols. The special symbols are endmarkers, which cannot be passed or overwritten. LBA are equivalent to Turing Machines whose amount of tape is restricted by some linear function of the input.

Context-free grammars are different from type 2 grammars in that all productions must have the form:

$$A -> a$$

where A is a single non-terminal symbol and a any string of terminal or non-terminal symbols. These grammars are recognized by push-down automata.

A context-free grammar, where all productions have either the form

$$A -> wB \quad \text{or} \quad A -> w$$

where A and B are variables and w a (possibly empty) string of terminal symbols, is called a right regular grammar. If all productions of a context free grammar have either the form

$$A -> Bw \quad \text{or} \quad A -> w$$

(A, B, and w have the same meaning as above) the grammar is called left regular. Left and right regular grammars characterize the regular sets, which are recognized by finite automata.

1.3. Transformational Grammmars

Transformational grammars are more powerful than phrase-structure grammars. The latter are not sufficient to handle all sentences in a correct way, as will be shown below. Sentences with embedded relative clauses or sentences with backward references cannot be handled by context sensitive rules alone. Noam Chomsky divides a sentence into its deep structure, surface structure, and semantic interpretation, all of which have to be treated in a different way. He suggests three different kinds of rules which make up the derivation rules for the grammar. In the first step of parsing an English sentence, phrase-structure rules are applied to it; in the next step, rules which can neither be regular nor context-free nor context-sensitive are applied. These rules therefore constitute the difference in power between phrase structure grammars and transformational grammars. The third and last part consists of morphophonemic rules. Morphophonemic rules are, for example:

take	-> [teyk]
past + take	-> [tuk]
past	-> [d]

These rules are tried in the order they are written.

1.4. Augmented Transition Networks

Augmented transition networks (ATNs) were introduced by William Woods [3]. ATNs have several less powerful predecessors such as transition networks and recursive transition networks.

An augmented transition network is a network of nodes and directed arcs connecting them. The nodes correspond to states in a finite state automaton and the arcs to the transitions from one state to another.

The power of a network is determined by what kind of labels for the arcs are allowed. The simplest labels consist of input symbols, i.e. terminal symbols of the underlying language. These are the labels allowed in a (simple) transition network. Transition networks are equivalent to regular grammars. In a transition network non-determinism is assumed. The occurrence of a specific symbol causes a certain transition from the state at the tail of the arc to the state at the head of the arc. For natural language processing this is not sufficient.

The above described network can be extended by introducing recursion. This extended network, called a recursive transition network, is a set of directed graphs consisting of states and arcs. One distinct state in each network is called the start state, another distinct set of states is called final or accepting states. The labels of the arcs may be names of networks or terminal symbols. Labelling the arcs with network names introduces the capabilities of a push down store. If a label which is a network name is encountered, processing proceeds as follows. The state at the end of the arc is saved in a push down store, and control is transferred to the start state of the specified network. This is done without advancing the input tape. Encountering a final state, the push down store is popped and control resumed by the state at the top of the stack. The attempt to pop an empty stack after the last input symbol leads to acceptance of the input. This makes the recursive transition network equivalent in power to context free grammars.

As pointed out earlier these are still not sufficient for dealing with natural language. Therefore augmented transition networks were introduced. They incorporate the features of the recursive transition networks but also allow tests and actions on a set of registers to be associated with arcs [3]. This gives ATNs the full power of Turing machines. In an ATN arcs from one state can lead to several successor states, since an ATN is a non-deterministic machine. It follows that some order in which the arcs are tried has to be defined. There are several ways of doing so.

For example a weight could be assigned to each arc. The arcs are then tried in descending order of weights. These weights could be changed during processing depending on certain already established features of a sentence. A similar approach is to associate a probability with each arc. Finally, the oldest and easiest way to define the order is to arrange the arcs in a definite sequence such that the program attempts each arc in that given sequence until it accomplishes a successful transition. This of course has the same effect as the two approaches mentioned above, if the arcs are ordered in decreasing probability. There is a difference however. In the former two cases the order can be changed, that is, the change of the order can be controlled by the program whereas in the last the order is static throughout execution. This scheme is sufficient for simple sentence structures and for cases in which the input does not change considerably in structure as in dealing with commands.

1.5. DIAGRAM, Systemic Grammars, Conceptual Dependency

1.5.1. DIAGRAM

DIAGRAM is a phrase structure grammar for a large subset of English. Its context free rules are augmented and give it limited context sensitive capabilities. DIAGRAM is an alternative to transformational grammars. J. J. Robinson distinguished four types of sentences and represents them separately [5]. These types are

imperative sentence (e.g. Put the apples in the basket!)

declarative sentence (e.g. He put the apples in the basket.)

propositional interrogative (e.g. Did he put the apples in the basket?)

argument interrogative (e.g. How many apples did he put in the basket?)

DIAGRAM is only slightly different from ATNs in that tDIAGRAM allows names of procedures as labels of arcs.

1.5.2. Systemic Grammar

Terry Winograd bases his system for dealing with the blocks world on a theory called systemic grammar [4]. One of the basic concepts of systemic grammar is the notion of syntactic units in the analysis of the structure of a sentence. He distinguishes three ranks of units in English: clause, group, and word. The largest unit is the clause. The reason why he does not choose a sentence as a unit, is that he considers sentences as units of discourse and semantics rather than syntactic units. Each of the units can be subdivided into several types, for example a group can be a noun group, a verb group, a preposition group, or an adjective group. The word is the basic unit and can have features. In other grammars for example the word "dogs" is distinguished from "dog" either as "dog" + plural or "dog" + "s". In systemic grammar, "dog" and "dogs" are considered basically the same with only a difference in a certain feature, the number. Whole groups can have features assigned to them. A noun group can be singular, plural, definite or indefinite and so on. One of the basic principles of systemic grammar is the rankshift. It means that any of the three ranks can appear anywhere in the structure tree. For example, a clause can be a part of a group, as in

"The man who came to dinner"

or part of other clauses, as in

"Join the Navy to see the world"(p.18 [4]). One could think that this makes the analysis more complicated. But certain features are only meaningful if others have been already established. It only makes sense to ask whether a sentence is a wh-question (like why, which,etc.) or a yes-no-question, if it is known already that the sentence is a question and not a declarative or imperative sentence. The realization rules in systemic grammar play a similar role as the transformation rules in transformational grammar.

1.5.3. Conceptual Dependency Theory

The conceptual dependency theory proposes that underlying natural language there exists a meaning structure [7]. If this structure is clearly defined, it should serve as a basis for all processes that use natural language input. The claim is that people think using some kind of meaning structure that is independent from the words of a language.

One principal rule is that any two sentences that have the same meaning have one and only one meaning representation underlying them regardless of their lexical form. Therefore it is necessary to search for primitive semantic elements, into which words with complicated meanings can be mapped. If this reduction is done correctly, it should be possible to explicitly state when words overlap or where they are equivalent in meaning. The goal is to define such a structure to use it for the representation of the output of a meaning analyser and as the basis of inference and memory programs.

MARGIE (Memory, Analysis, Response Generation, and Inference in English) is a set of three programs to understand natural language based on the conceptual dependency system. These three programs are:

1. analysis program
2. memory program
3. generator.

The analysis program maps sentences into conceptual structures. From these structures the memory program makes inferences and the generator finally codes conceptual structures back into English. These programs together function as a paraphrase and inference system.

In [6] R. Schank describes the program SAM (Script Applier Mechanism), which uses scripts to make inferences in a domain it knows about. The Program was added to MARGIE.

A script is a structure that describes an appropriate sequence of events in a particular context, i.e. a predetermined, stereotyped sequence of actions that define a well-known situation. A script consists of slots. Certain requirements with respect to the contents of these slots have to be fulfilled. A script is an interconnected structure, i.e. the contents of one slot affects what contents is allowed in another. Verbs are divided into several actions, which are described by ACTs of the conceptual dependency theory. For example the action 'propelling' covers the verbs like pushing, throwing (propelling something or someone) or running, walking (propelling oneself). Each action results in conditions that enable the next action to occur. This is called the principle of causal chaining. An action has to be completed satisfactorily before the next action in the sequence can be performed. The completion of an action can be prevented by either an obstacle or an error. In either case a what-if ACT is called, i.e. a new action in the script will be generated in order to get things moving again.

When a new script is being learned, every time an error or obstacle is encountered, ways to remove this obstacle or correct that error are stored in a what-if ACT. The restaurant script serves as an example. It is divided into several scenes, like entering, ordering, eating, exiting. Each scene has ACTs associated with it. The scene 'exiting' has among others the following ACTs:

ATRANS - receive check

ATRANS - tip waitress.

SAM receives conceptual dependency structures as input, identifies the script to be used, and fills this script with the new input as it is received. The structure resulting from this can be used to make summaries or paraphrases of the original story. The generation program maps the answers of paraphrases back into English. In the example above, assume the customer is not satisfied with the service and does not leave a tip. The program SAM infers that the amount of money left as a tip is in direct correlation with the degree of satisfaction of the customer. This dependence is not explicitly stated in the script.

Plans are responsible for the deliberate behavior people exhibit. They describe the choices a person has to achieve a goal. Seemingly independent, disconnected sentences are often related to the achievement of goals of the same plan. Discovering the underlying plan allows a person to make sense of a story.

Since we do not have scripts for every possible story, general goal states have to be recognized. For example, once a general goal like 'raising a lot of money' is recognized, the specific purpose like paying for a house, or financing a child's education is of minor importance.

In order to achieve a goal very often an intermediate goal has to be achieved first. In the above example 'contacting a wealthy relative' could be such an intermediate goal. Special ACTs, so called delta-ACTs are defined for this purpose.

A sufficiently large number of scripts has to be available in order to use SAM for a variety of scenarioes.

1.6. Comparison of Methods

Comparisons of the tools for parsing natural language sentences are in the following always based, as stressed by some authors, on the assumption that the natural language dealt with is English. For other natural languages different approaches might be more profitable due to grammatical idiosyncracies of each language.

Madeleine Bates in [2] explains ATNs using as an example the program LUNAR. LUNAR is a question answering system that answers questions about rock samples brought back to earth from the moon, and was one of the first major applications of ATNs.

Madeleine Bates compares several approaches to language processing and concludes that ATN's are the best choice for parsing natural language sentences. Since ATNs have changeable registers and can transfer control depending on the state of those registers, they are equivalent to Turing Machines and therefore more powerful than transformational grammars. Although not all of their power is actually needed for parsing natural language, they are more convenient to write and use than transformational grammars. The set of rules needed for a transformational grammar can become very complex and difficult to comprehend.

In comparison to transformational grammars ATNs also have considerable advantages in regard to execution. Firstly, transformational grammars are appropriate only to the generation of sentences. The generation of sentences is performed by first constructing the deep structure of a sentence by means of a context free grammar and then transforming this structure into a surface structure by successive application of transformations. The leaves of the tree that represents the structure constitute the final form of the sentence. The transformational grammar approach includes algorithms for analyzing sentences, too. But these algorithms are too inefficient to be considered for practical application. The original algorithm performs analysis by synthesis. This means that the rules for the generation of a sentence are applied for all possible sentences. While generating them, it is checked, whether the sentence to be analyzed is among the sentences generated so far. Developing more efficient algorithms leads to two different categories. The algorithms are either too time consuming for the analysis of a large set of sentences or are formally incomplete. ATNs on the other hand constitute a dual model. They are well suited for the generation as well as the analysis of sentences. Another reason why, according to Woods, ATNs are preferable to transformational grammars is that the transition networks derived from a transformational grammar by a mechanical procedure are not as efficient as the transition networks developed by hand.

The difference between the transformational and systemic grammar can be shown by a simple example:

"Sally saw the squirrel."

"The squirrel was seen by Sally."

"Did Sally see the squirrel?" (p.21, [4])

The three sentences have a different structure in systemic grammar. In the transformational grammar, these sentences have an almost identical deep structure and are differentiated only by the tranformation rules.

T. Winograd also compares his system to the ATN approach. He defies Woods' opinion that ATNs are easy to read in comparison to transformational grammars. Winograd states that for a complete language ATNs become very complex, too. PROGRAMMAR is a programming language, implemented in LISP, that is based on systemic grammars and facilitates natural language processing. According to Winograd there is no difference in power between PROGRAMMAR and ATNs. It is easy to translate the notation of ATNs into a program, which then would be similar to a PROGRAMMAR program for a systemic grammar. The difference lies in the types of analysis carried out by the two systems. Network systems try to reproduce the deep structure of a sentence while doing surface structure recognition. But since PROGRAMMAR is based on systemic grammar, it identifies significant features in the constituents being parsed. To add additional power to the basic parser, PROGRAMMAR has demons. This interrupt mechanism could be added also to ATNs by special labels from each arc with a name corresponding to each demon in PROGRAMMAR. The third difference is the backup-mechanism. Networks generally assume nondeterminism and therefore have an automatic back-up. A more 'intelligent' parser on the other hand 'understands' the reason for a failure, and backs up accordingly instead of blindly. This is the way it is done in PROGRAMMAR. One of the inefficiencies of transformational grammars consists in the fact that the syntactic tree structures first have to be built and then transformed. After the transformation some parts may be deleted, which makes it redundant to build them in the first place. J. J. Robinson tries to avoid this.

Variations of the ATN model are the Thorne system and the system by Bobrow and Fraser. The former tries to represent simultaneously the deep structure and the surface structure of a sentence. The latter is an elaboration of the former. According to Bobrow the main differences between the two approaches are a facility for mnemonic state names and the ability to transfer information back to previously analysed constituents. Another difference is that subroutines can be stored directly in the dictionary entry of a word instead of merely being activated by the features stored in the dictionary.

Altogether ATNs have to be considered the state of the art in natural language processing.

2. Program and ATN Descriptions

2.1. High-level descriptions of procedures and data structures

The purpose of this program is to process natural language sentences which represent editor commands. These commands are aimed at manipulating a textfile or document segmented into paragraphs. The execution of the editor commands is simulated and the list of syntactically correct commands to be used for interacting with the QED editor on the Honeywell system of KU is returned.

I first want to explain on a high level what the program does as far as it is visible to the user. The discussion of how it works will follow afterwards.

In the interaction with this interface the user is asked for the name of the file she wants to work with. In order to simulate the execution of the commands further information concerning the number and lengths of the paragraphs is needed. Then the user can begin to input the commands. If for any reason the command fails to be executed, the user is informed about the reason for rejection and any executable part of the command is ignored. In either case, whether the command is accepted or not, the user is asked for the next command. The work on one file can be finished by entering the empty list -()- instead of a command. The user is then supplied with information about the updated file, i.e. the number and lengths of the paragraphs after the simulation of execution. Also the list of QED commands is given. The user now has the option of working on another file by giving its name, in which case the cycle outlined above is repeated. By again entering the empty list -()-, the user indicates that she wants to quit. In the latter case the user exits the program.

Two data structures available in LISP besides atoms and constructed lists are used: property lists and association lists. Properties are used to associate features of a file to its name; an association list is used to relate the number of a paragraph with its length.

The following assumptions concerning the structure of a text-file are being made:

1. Each paragraph except the last one is followed by exactly one blank line.
2. The first line of the file is not a blank line.
3. If there is a title, it is followed by exactly two blank lines.

In my program I make the assumption that the user has a hardcopy of the document she works with. Referring to this hardcopy which is the original version at the beginning of the session, she makes her changes. Therefore I assume in the numbering she always refers to the original.

The user is asked for the name of the file she intends to work with. This name becomes the value of the variable filename. When the user is asked to supply information about the file she intends to work with, this information is stored on the property list of the value of the variable filename. The number of paragraphs is stored under the property nr_of_par. The length of each paragraph is stored in an association list under the property par. The value of the variable filename remains unchanged the whole time the user is working on one specific file. A second variable, called #newfile has a value derived from the value of the variable filename. If, for example, the variable filename has the value 'f2', #newfile has the value '#f2#' Associated with the value of #newfile are the same property lists as with the value of filename.

There are two additional properties associated with the value of #newfile, curr_line and curr_par, which keep track of the line and paragraph referred to most recently (with regard to the original file). Initially they are both set to 0. After the simulation of each command these values are altered to reflect the changes that have occurred in the file. For example adding or inserting (or deleting) a line in paragraph x --where x is a positive integer between 1 and the number of paragraphs-- results in an increase (decrease) of the value for x in the association list under the property par.

Furthermore two variables are necessary to record the exact changes to the file during the program: ch_par for paragraphs, ch_line for lines. They both have the properties a, i, and d for append, insert, and delete respectively. The values of the properties are lists of absolute line and paragraph numbers with respect the original file. The elements of these lists are in ascending numerical order. If a line is deleted, its number with respect to the original file is added to the list under the property d of the variable ch_line while preserving the ascending numerical order of that list. If the only remaining line of a paragraph is deleted, not only is the value of the property d of ch_line changed, but the value of the same property of ch_par is changed accordingly. In that case the value of the property nr_of_par of the value of #newfile is also decreased by 1. If a paragraph is deleted, its number is added to the list under the property d of the variable ch_par. Also the numbers of all lines in the deleted paragraph are added to the list under the property d of ch_line.

There are no special properties to record the effects of move or copy commands. If a line is moved, its number is added to the list under the property d of ch_line. If it is moved before (after) a specific line, the number of this line is added to the list under the property i (a) of ch_line. Analogous changes are made if paragraphs are moved. If a line/paragraph is copied, it is not removed from its original location. Therefore the only changes recorded are adding the line/paragraph number of the new location to the lists under the properties i or a depending on whether the line/paragraph is copied before or after that location.

Assuming a file with name f1, no title, and 4 paragraphs with the lengths of 4, 13, 2, 7 lines respectively, the variables have the following values:

variable	property	value
f1	nr_of_par	4
	par	((1 4)(2 13)(3 2)(4 7))
#f1#	nr_of_par	4
	par	((1 4)(2 13)(3 2)(4 7))
	curr_line	0
	curr_par	0
ch_par	a	nil
	i	nil
	d	nil
ch_line	a	nil
	i	nil
	d	nil

Assuming that line 3 in paragraph 4 (absolute line number 25) is deleted, the variables change their values as follows:

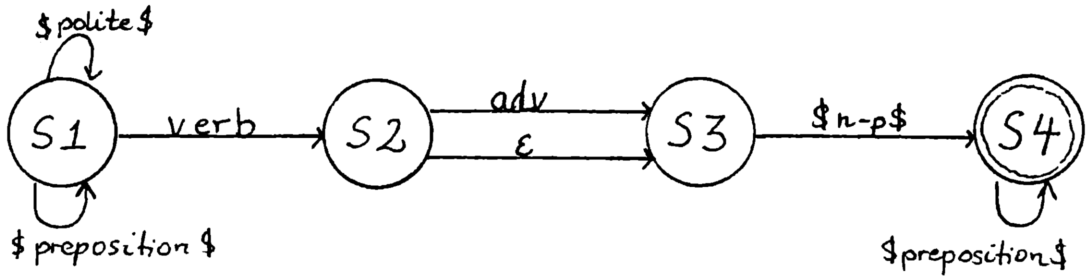
variable	property	value
f1	nr_of_par	4
	par	((1 4)(2 13)(3 2)(4 7))
#f1#	nr_of_par	4
	par	((1 4)(2 13)(3 2)(4 6))
	curr_line	25
	curr_par	4
ch_par	a	nil
	i	nil
	d	nil
ch_line	a	nil
	i	nil
	d	25

After all necessary information about the file is gathered, the program is ready to accept commands from the user. The command a user wants to be executed has to go through different stages of processing before it is transformed into a proper QED command (Fig. 2.1.).

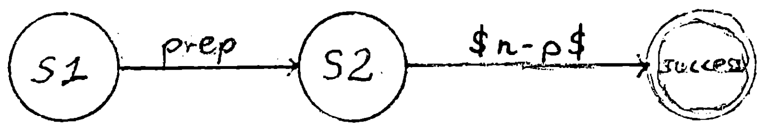
$$\begin{tikzcd}
\text{INPUT} \arrow[r] & \boxed{\text{prepare-input}} \arrow[r] & \boxed{\text{ATNs}} \arrow[r] & {}
\end{tikzcd}$$

$$\begin{tikzcd}
 & & \boxed{\text{split}} \arrow[dr] & & \\
{} \arrow[r] & {} \arrow[ur] \arrow[dr] & & \boxed{\text{comm - distr}} \arrow[r] & \text{QED command} \\
 & & \boxed{\text{moving}} \arrow[ur] & &
\end{tikzcd}$$

FIG. 2.1.



$ command $



$ preposition $

FIG. 2.2.

The command is first preprocessed in a function called prepare-input. This function replaces text constants by the dollar sign ($) and numbers by the number sign (#). The actual values of texts and numbers are stored in two separate variables text_list and num_list. With a number is stored an indication of whether it is a cardinal or an ordinal number. Cardinal and ordinal numbers have to be distinguished. If an ordinal number occurs that means the noun it belongs to follows it whereas a cardinal number is preceeded by its noun "paragraph 2" versus "2nd paragraph".
Therefore a phrase like

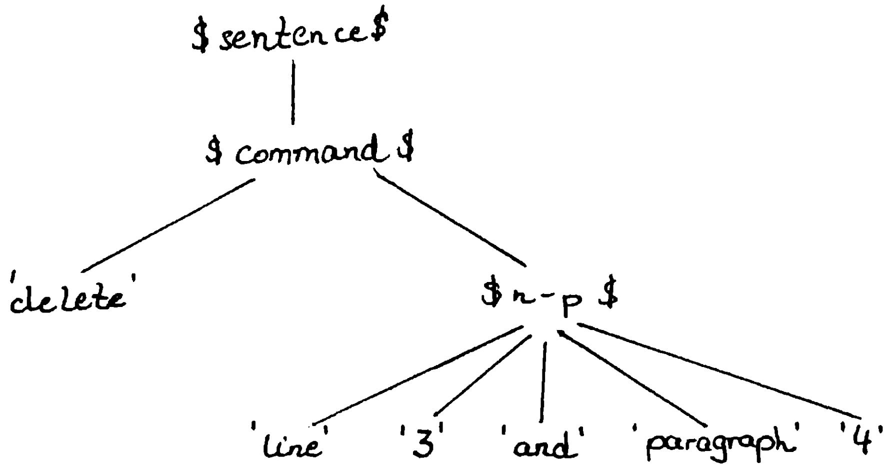
the 2 line in the 1st paragraph

is not accepted (singular and plural are not distinguished). Expressions like "...5 to 7..." "...5 7 8 10..." are replaced by "...5 and 6 and 7..." "...5 and 7 and 8 and 10..." respectively.

The preprocessed input is then subject to the ATNs and the ATN compiler as written by Dr. Albert D. Bethke.

Sentences are accepted or rejected according to the structure of the ATNs (see Fig. 2.2., and Appendix A) and the entries in the dictionary. Since the structure of sentences dealt with is not very complicated, the networks themselves are not very complex.

delete line 3 and paragraph 4



delete line 3 in paragraph 4

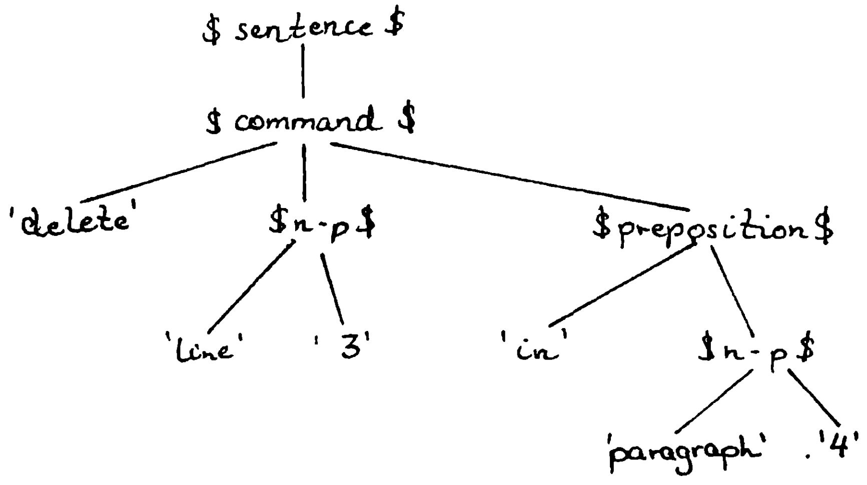


FIG. 2.3.

The ATN compiler requires an ATN named $sentence$. The ATN $command$ accepts declarative sentences. A sentence can begin with an adverbial phrase as "would you please...". In this case control is transferred to the ATN $polite$. It is also acceptable to start a sentence with a prepositional phrase like "in paragraph 3...". Then the ATN $preposition$ is called. Thirdly, a sentence can begin right with the command, e.g. "delete   Either of the foregoing cases can follow each other in different ways. For example:

"would you please in paragraph 3 delete line 4"

"delete please in paragraph 3 line 4". After the kind of command is established and extracted, $n-p$ is called. $n-p$ is traversed successfully, if

1. an ordinal number followed by an appropriate noun
   or a sequence thereof,
2. a noun followed by a cardinal number
   or a sequence thereof,
3. an adverb followed by a cardinal number followed
   by a noun
   or a sequence thereof

is encountered. If the sentence is exhausted on leaving $n-p$, the sentence is accepted. Otherwise $preposition$ is called possibly several times. The network $preposition$ is successfully traversed if the input consists of a preposition followed by a noun phrase.

The structure of the ATNs allows the return of different structures the the calling program depending on the relationships between noun groups. In the case of a phrase like "...line 3 and paragraph 4..." $n-p$ is called and the reference to the line and to the paragraph are both handled by this network, whereas in the case of "...line 3 in paragraph 4..." after $n-p$ is successfully traversed and control is given back to $command$, the latter calls $preposition$ which again calls $n-p$ (Fig. 2.3.). If a sentence is accepted, some information about the structure of the command is stored in a global variable. The ATNs are built so as to detect which number(s) belong to which nouns. For example: "move line 3 and 5 after line 4 in paragraph 3" must be distinguished from "move line 3 after line 4 and 5 in paragraph 3". Furthermore "delete line 3 in paragraph 4" is different from "delete line 3 and paragraph 4". The function split gets as input the list that is passed from the ATNs. It is responsible for partitioning the command into a sequence of commands, if necessary. If the command includes references to several distinct locations (except references like ...last 3...) it has to be split since only one location can be handled at a time. So for each location referred to a command is assembled with the denotator for the command, whether it is a paragraph or a line, and the number. Several structures of sentences have to be distinguished. In the following the different structures will be given by examples which state the inputs or outputs from certain stages of the processing.

Example 1

Input from user: (delete line 3 and 4)

Output from ATNs: (line 3) (line 4))

Output from split: (delete line 3) (delete line 4)).

A sentence like (delete the last 3 lines) on the other hand is transformed into (delete line last 3).

Example 2

Input from user: (delete line 3 in paragraph 4)

Output from ATNs: ((line 3) (paragraph 4))

Output from split: (d line 3 paragraph 4)

Example 3

Input from user: (delete line 3 and 4 in paragraph 6)

Output from ATNs: ((line 3 4) (paragraph 6))

Output from split: ((d line 3 paragraph 6) (d line 4 paragraph 6))

In this case the function split detects that more than one line is referred to. An analogous transformation is performed if several paragraphs are referred to.

Example 4

Input from user: (delete line 3 in paragraph 4 and 5)

Output from ATNs: ((line 3)(paragraph 4 5))

Output from split: ((d line 3 paragraph 4)(d line 3 paragraph 5))

Example 5

The case of references to more than one line in more than one paragraph is structurally the most complicated.

Input from user: (delete line 3 and 4 in paragraph 6 and 7)

Output from ATNs: ((line 3 4)(paragraph 6 7))

Output from split: ((d line 3 paragraph 6)(d line 4 paragraph 6)(d line 3 paragraph 6) (d line 4 paragraph 7)).

Since in sentences beginning with a prepositional phrase the order in which lines and paragraphs appear is reversed, the program checks whether a paragraph is referred to prior to a line. If that happens, the sequence of references to lines and paragraphs in the list returned by the ATNs is reversed and in the following treated like the sequences above. The transformation of 'move' and 'copy' commands has to be different from the one above. In the ATNs the occurrence of a word like 'after' or 'before' is checked in order to distinguish the references to the item to be moved/copied and to the location of the moving/copying. It is marked with a appended to the list as it is built at the moment of the encounter. Since it is important to know whether the item is supposed to be moved/copied before or after the location mentioned, the name of the command is slightly changed in order to convey this information. In the case of moving/copying after (before) the location, the commands 'm' and 'k' are changed into 'ma' ('mi') and 'ka' ('ki') repectively. The function 'moving' rather than 'split' is called. Here again we have to distinguish several cases as for all other commands.

1. Input from user: (move line 4 after line 7)

   Output from ATNs: ((line 4)*(line 7))

Because of the simplicity of this sentence, in moving no changes are made to the format.

   Output from moving: (ma (line 4)(line 7))

2. Input from user: (move line 4 in paragraph 3 after line 7)

   Output from ATNs: ((line 4 paragraph 3)*(line 7)

   Output from moving: (ma (line 4 paragraph 3) (line 7))

3. Input from user: (move line 3 and paragraph 3 after line 7)

   Output from ATNs: ((line 4)(paragraph 3)*(line 7))

It is obvious that the reference to the paragraph is only connected to the first item in the list, i.e. that 2 items are to be moved to one location, and not one item to two different locations.

   Output from moving: ((ma (line 4)(line 7))(ma (paragraph 3) (line 7)))

4. Input from user: (move line 3 after line 4 and 5)

   Output from ATNs: ((line3)*(line 4)(line 5))

If an item is to be moved to several new locations, the command is transformed in such a way that it is moved only to the last mentioned location and copied to the others. This is necessary, as discussed above, for moving an item is followed by deleting it from its original location. An attempt to move the same item again leads to a message that this item had been deleted before.

If this were transformed into

((ma (line 3)(line 4)) (ma (line 3)(line 5))) this command could not be executed. In the simulation of a move command, the item to be moved is deleted at the original location. So after executing the first part of the sequence built above, line 3 does not exist any more and the execution of the second part would lead to a message to the user saying that line 3 had been deleted before. Therefore

Output from moving: ((ka (line 3)(line4))(ma (line 3)(line 5)))

No problems arise with this sequence, since the 'copy' command leaves line 3 untouched.

Also during further processing the commands to move or copy single lines or whole paragraphs have to be treated differently from the commands discussed earlier. The reason is that their format differs from the format of the other commands. In these commands two sorts of information are incorporated, one refers to the 'object' to be moved or copied (the original location), the other to the location which the object is to be moved/copied to (the new location). So the new location has to be extracted from the command. The whole sequence of functions is called as described earlier to determine the actual location, where the object is. Then the same is repeated for the original location. In the case of the move command all lists are updated by simulating the deletion of the item from the original location and its addition at the new location.

In the function comm-distr the provisions for correct output according to different commands are made.

In the following I first want to discuss how commands such as 'print', 'delete', 'insert', 'append', and 'substitute' are created. The discussion for 'move', 'copy', and 'exchange' will follow separately since the structure of these commands differs considerably from those mentioned before. The 'exchange' command is split into the sequence consisting of the commands to move the first item after the second, then the second after the original location of the first and to delete both items at their original places.

There are basically five types of clauses to be distinguished, the simplest consisting of commands which refer to a specific line or paragraph number explicitly. These are commands containing phrases like ...line 4..., ...paragraph 3..., ...the third line..., ...the 4th paragraph... These sentences can be passed on for further processing without any changes. Equally simple and belonging to the same category of sentences are references like ...last line..., ...first paragraph..., ...next line..., ...previous paragraph... The adverbs last, first etc. are replaced by the appropriate values; e.g., in the case of last line the function get-dollar is called, which calculates its absolute line number from the given information about the number of paragraphs and their lengths. Two variables exist in which the line and paragraph numbers of the line and paragraph most recently referred to are stored. They are initialized to 0, so that at the beginning of a session next line/paragraph and first line/paragraph are synonymous. In later processing it is checked whether reference to their values is possible. There are two possible reasons that a command cannot be executed. The first is that the line or paragraph referred to has previously been deleted. The second is that that line or paragraph does not exist at all; e.g., reference to line 100 if the file only contains 90 lines. In these cases the user is prompted with a message explaining why the command cannot be executed. Otherwise, it is checked whether lines or paragraphs with lower number have been added, inserted or deleted and the line or paragraph respectively is adjusted so that it refers to the correct line/paragraph in the updated file. I.e., for each line/paragraph added or inserted the value of the line/paragraph currently under consideration is increased by 1; for each deleted line/paragraph it is decreased by 1. This is done for lines and paragraphs in separate functions. One value is then returned and in comm-distr the syntactically correct command is assembled. The lists for added, inserted and deleted lines/paragraphs are updated after the complete command is assembled.

In the next category belong commands with references like ...first 4 lines..., ...previous 3 paragraphs..., ...next 2 lines..., ...last 5 paragraphs... Here a function is called that calculates the first and last line/paragraph referred to; e.g., first 4 lines returns (1 4). All following functions are called twice, once with the lower value, once with the higher value of the list.

In category three there are references such as ...line 3 in paragraph 4..., ...the first line in the last paragraph..., ...the fourth line in the previous paragraph..., etc. I choose as an example for this category (delete line 3 in paragraph 4) This sentence is transformed into (d line 3 paragraph 4). Then the absolute line number with reference to the original file is calculated. It is checked whether paragrah 4 exists in the file. If that is successful, it is checked whether line 3 exists in paragraph 4. After that as in the above cases the line number with reference to the updated file is calculated and messages given to the user, if the line has been deleted before.

The next category has to be divided into two subcategories. The first contains sentences like ...line 4 in the last 3 paragraphs... Sentences of the first subcategory are treated the following way. A list of paragraph numbers is built, e.g. if the document that is being worked with has six paragraphs, the above ex-

ample would lead to the list (4,5,6). The further processing is then done for the fourth line of each paragraph (4,5, and 6) separately. In the second subcategory we find sentences like ...last 3 lines in paragraph 4... . Here a list of two elements is constructed. The elements of the list are the numbers of the first and last line referred to, in the example given the list would be (5 7). Then the line numbers are calculated as if the original command had been the sequence
((delete line 5 in paragraph 4)(delete line 7 in paragraph 4)).
The difference between a sequence of commands of category 3 and the above is that in category 3 the commands would be treated as two separate commands, whereas here the result of the two commands are combined into a single command.

The last category contains references like ...last 3 lines in the first 4 paragraphs... . Here the methods applied to the two types of category four are combined. The paragraph numbers of the paragraphs referred to are calculated and put into a list, in the example (1 2 3 4), whereas for the lines only the first and last line referred to are mentioned. Then the commands are processed for each paragraph separately the way it is done in the second subcategory of the previous category.

2.2. Justification for the approach chosen

The reason why I chose ATNs over transformational grammars is threefold. ATNs are, for one thing, easier to deal with than transformational grammars; they are easier to read and comprehend. Secondly, my sentences are structurally and grammatically very simple, in that all sentences are statements in the present tense, no minor clauses and attributes are necessary. As mentioned in section 1, the purpose of the program is to analyze sentences and not to generate them. If the latter had been the purpose of the program, transformational grammars would have probably been as appropriate as ATNs; but for the analysis of sentences ATNs are superior to transformational grammars.

3. Performance and Limitations, Suggestions for further work

The program is capable of simulating the execution of a variety of commands. The dictionary entries allow the user to formulate the command in different ways. For example, the user can use delete, remove, eliminate as synonyms. Furthermore it is permissible to use adverbial phrases like ...would you..., ...please...., ...furthermore... to suit the user in her desire to conduct a natural conversation. The user has three choices of referring to a specific line. She either refers to it explicitly by giving its absolute line number as in ...line 23..., or in terms of the previously referenced line, as in ...next line... In addition she can refer to it in relation to a paragraph as in ...line 3 in paragraph 4... The first two cases are applicable similarly for references to paragraphs.

Most of the restrictions imposed on the form of the input stem from the fact that QED does not allow interaction directed from a program. Therefore the commands cannot be executed but the execution can only be simulated. This is done by using the information about the original file and keeping information about the manipulations of this file. The number of lines and paragraphs being deleted, appended, inserted, moved, or copied is stored and based on this information line and paragraph numbers are calculated to reflect the structure of the file as if the (preceeding) commands had actually been executed. It would be very convenient to store information on user files in a separate datafile or connected with the files themselves. Then the user would be asked solely for the name of the file and all necessary information could be retrieved by the program. Since this is not possible, the user has to be asked to enter number and lengths of the paragraphs explicitly.

Since in LISP every input is an s-expression, the texts in the substitute, append and insert commands have to be surrounded by (double) quote marks ("). The LISP interpreter reduces sequences of blanks to exactly one blank, so expressions that involve more than one blank in sequence cannot be distinguished unless they are treated as a text constant. That also implies that the double quotes have to be output, which is, at the least, a nuisance for reading the command. Furthermore, the period has a special meaning in LISP, denoting the dotted pair, and therefore cannot be used in input sentences. It is impossible to reflect the exact format of the substitute command using LISP alone. If the text constant contains symbols that are treated specially in QED like $, *, [, etc. it is impossible to switch off the special meaning and assume the literal meaning. To switch to the literal meaning, the special symbol would have to be preceeded by \c. It is not possible to manipulate the text constant so as to insert \c. This is especially awkward in the case of the period. Its special meaning is that it matches any character. If the user intends to change a period into a comma for example, the input is transformed into s/"."/","/p. If this is used in the form above, with just the (double) quote marks stripped off, the result would be s/./,/p. But this is not what the user intended. This command converts every character into a comma. To perform what the user intended, the command should have the format s/\c./,/p. It would be nice to have an interface between the LISP program and the editor, perhaps in the form of a PASCAL program that could take care of these things.

Although the ATNs themselves are able to handle a broader variety of commands and only slight changes in the rest of the program would expand the power of the program, again QED poses restrictions on the choice of permissable commands. After expanding the dictionary accordingly, a command like (find "this sequence") could be parsed successfully and transformed into /"this sequence"/. After simulating the execution it would be impossible to assign the correct values to the variables holding the current line and current paragraph numbers. It is not even possible to determine whether the context search was successful or not. If future commands refer to a line with a certain context, it is impossible to determine whether that line has been deleted. Updating information about the file is therefore made impossible. Example: assume line 3 has been deleted before and (find "this sequence"), transformed into /"this sequence"/, refers to line 3. In the actual execution of the command this occurrence of the text has to be skipped (it does not exist any more) and it has to be searched repeatedly. Another example: assume the command sequence

(delete line 3)

(find "this sequence")

(delete the next line)

Furthermore assume the context search leads to line 2. Then the last command in the above sequence would have to result in the message "Line 3 has been previously deleted. The command cannot be executed.".

But since in the simulation it is not possible to receive line numbers from the editor, it is impossible to allow the context search. Analogous reasons apply for forbidding phrases like (substitute the first occurrence of "This" by "That") It is impossible to determine the line and paragraph number of the first occurrence of the text, if it appears in the file at all. In a system that allows direct interactions the above mentioned problems would immediately cease to exist. One way to deal with context search is to pass the message from the editor to the user if the search fails and otherwise determine the line number by a command equivalent to the "=" in QED and use the returned value for further reference.

It is necessary to impose some restrictions on the format of the document, concerning the structure of paragraphs. Some possibilities are not allowing blank lines within a paragraph, or requiring markers for the beginning of paragraphs. Another way would be to require indentation of the first line in a paragraph or a header "paragraph x" before each paragraph. In my program, exactly one blank line between paragraphs is assumed. In my case blank lines in a paragraph are treated the same way as lines that contain text. The only entities my program deals with are lines and paragraphs. It is not possible to refer to specific sections within a paragraph, therefore section and paragraph are treated as synonyms. Problems occur also with syntactical ambiguities caused by the fact that only one command at a time can be simulated. The user has a different intention when she types (move paragraphs 3 and 4 after paragraph 7) than when she types (move paragraph 3 after paragraph 7) (move paragraph 4 after paragraph 7). In the former case paragraphs 3 and 4 are together considered a single entity and the new sequence of paragraphs would be ...7,3,4... whereas in the latter case the execution of the first command results in the sequence ...7,3,... and the following execution of the second command leads to ...7,4,3,... I decided to treat both cases in a uniform way. Since my program splits the first example into two distinct commands, which are the same as the sequence of the second example, I decided in favor of the latter alternative.

In a refinement of this program either both cases could be treated differently or the user could be asked to specify the sequence uniquely. E.g. if she wants the sequence 7,3,4 she should type (move paragraph 4 after paragraph 7)(move paragraph 3 after paragraph 7), and if she wants 7,4,3 she has to enter (move paragraph 3 after paragraph 7) (move paragraph 4 after paragraph 7). As pointed out earlier, in my program I make the assumption that the user has a hardcopy of the document she works with. Referring to this hardcopy which is the original version at the beginning of the session, she makes her changes. Therefore I assume in the numbering she always refers to the original. I consider this a justified point of view, because having a written version it is easier for the user to refer to it than to keep track of all the changes she has made before, especially adding, deleting, inserting, moving, or copying lines or paragraphs and therefore calculating the line/paragraph numbers she is referring to for the up-to-date version of her document. In an interactive mode with the editor, when the changes are made immediately -online- we would assume that the user refers to the current version of the document, and there would be no need for all the calculations my program does in order to deduce the actual line/paragraph referred to in the current version from the original. A big part of my program would then be redundant, because the work could be done by the editor directly. For further refinement of the program more than one command could be allowed at a time. In my program a phrase like

(in line 3 replace " by " and delete the next line)

is not accepted. In an extended version this sentence could be taken apart to form two separate commands (by some mechanism relating the verbs to the appropriate objects) which are then executed (in sequence).

REFERENCES

[1] Noam Chomsky : Structural Grammars, 1957

[2] Madeleine Bates : The Theory and Practice of Augmented Transition Network Grammars,
in: Natural Language Communication with Computers
p. 190-260 Berlin, Springer Verlag, 1978

[3] W.A.Woods: Transition Network Grammars for Natural Language Analysis
CACM 13 pp.591-606 1970

[4] T. Winograd:Understanding Natural Language
Academic Press New York and London, 1976

[5] Jane J. Robinson : DIAGRAM : A Grammar for Dialogues
CACM 25,1 pp. 27-47 Jan. 1982

[6] R. C. Schank and R. P. Abelson :
Scripts, Plans, Goals and Understanding, 1977
Hillsdal, N.J.

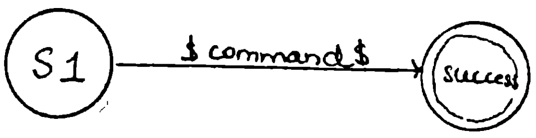
[7] R. C. Schank et al. :
Inference and Paraphrase by Computer
Journal of the ACM 22(3), p.309-328, 1975

[8] John E. Hopcroft and Jeffrey D. Ullman :
Introduction to Automata, Theory, Languages
and Computation, Chapter 9,
Addison Wesley 1979, pp. 217-227

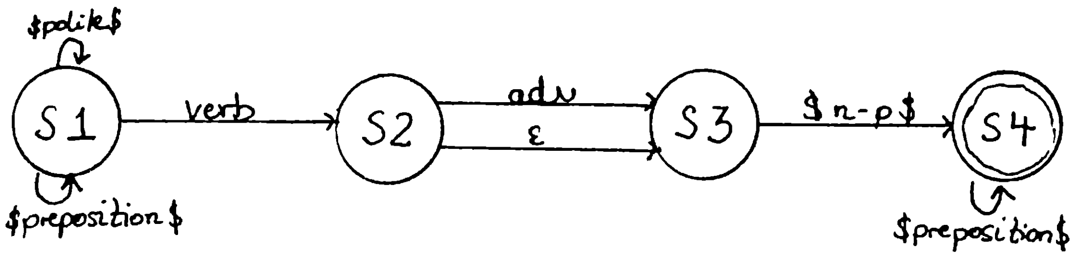
APPENDICES

APPENDIX A

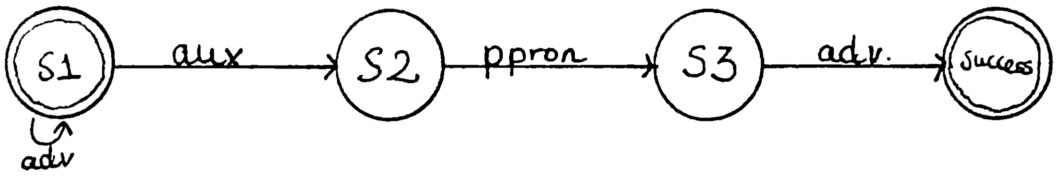
A T N s



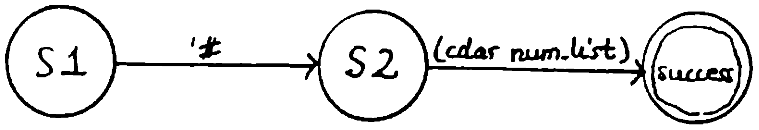
$ sentence $



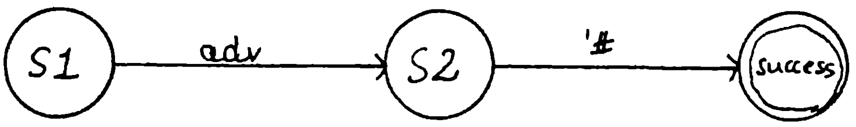
$ command $



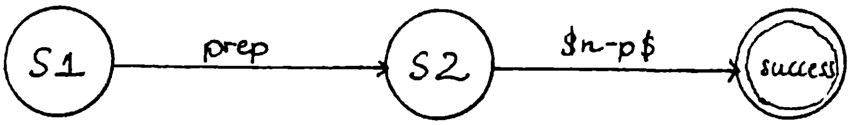
$ polite $



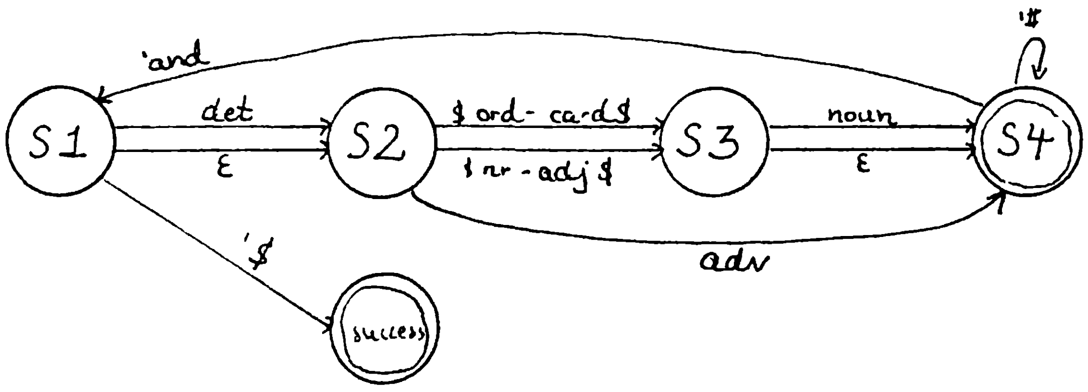
$ ord-nr $



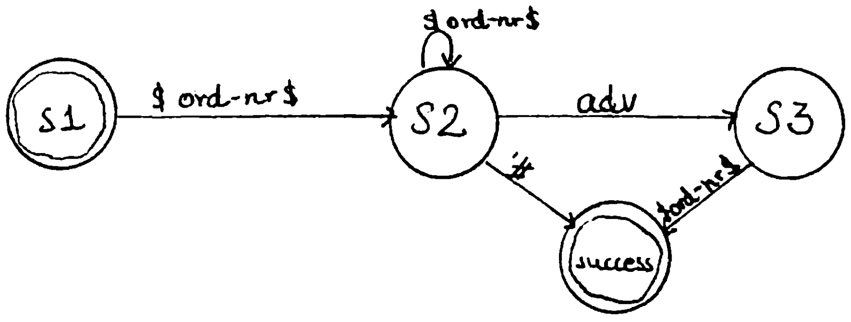
$ ord - card $



$ preposition $



$ n- p $



$ nr- adj $

```
(define-atn
     $sentence$ ()
 (s1 (if $command$
               go success
     )
 )
)%$sentence$


(define-atn
   $command$  (fix fixuplist qed-com kind check_later f1 ordv fix )
 (s1 (if (and $word$ (member 'verb $p-o-s))
     go s2
       after
         (setq comm $command)
     )
     (if $preposition$
     go s1
       after
         (cond (f1
                   (setq f2 (append f2 f1))
                   (setq f1 nil)
               )
               (t
          (setq f2 (append f2 (list (append  (list kind) fix))))
          (setq fix nil)
               )
         )
         (setq kind nil)
     )
     (if $polite$
     go s1
     )
 )
 (s2 (if (and  $word$ (member 'adv $p-o-s))
     go s3
     )
    (else go s3)
 )
 (s3 (if $n-p$
     go s4
       after
          (cond (f1
                   (setq f2 (append f2 (list (list kind f1))))
                   (setq f1 nil)
              )
              (t
                (setq f2 (append f2 (list (append (list kind) fix))))
                (setq fix nil)
               )
        )
       )
)
(s4 (if $preposition$
    go s4
      after
         (cond (f1
                (setq f2 (append f2  f1))
                (setq f1 nil)
          )
          (t (setq f2 (append f2  (list kind fix)))
             (setq fix nil)
          )
        )
    )
    (else go success
            after
               (while check_later
                          (setq check_l (car check_later))
                          (cond ((and (equal check_l 'after)
                                      (equal comm 'i))
                                     (setq comm 'a)
                                )
                                ((and (equal check_l 'before)
                                      (equal comm 'a))
                                    (setq comm 'i)
                                )
                                ((and (equal check_l 'into)
                                      (equal comm 'k))
                                    (setq comm 's)
                                )
                                ((or (equal comm 'm) (equal comm 'k))
                                      (cond ((equal check_l 'before)
                                              (setq comm (compress
                                                 (list comm 'i)))
                                            )
                                            (t (setq comm (compress
                                                  (list comm 'a)))
                                            )
                                     )
                                 )
                                 (t nil)
                         )
                (setq check_later (cdr check_later))
               )%while
  )
)
```

```
)%command


(define-atn
   $polite$ ()
 (s1 (if (and $word$ (member 'adv $p-o-s))
     go s1
     )
     (if (and $word$ (member 'aux $p-o-s))
     go s2
     )
     (else go success)
 )
 (s2 (if (and $word$ (member 'ppron $p-o-s))
     go s3
     )
 )
 (s3 (if (and $word$ (member 'adv $p-o-s))
     go success
     )
     (else go success
     )
 )
)%polite


(define-atn
  $nr-adj$ ()
 (s1 (if $ord-nr$
     go s2
     )
     (else go success)
 )%s1
 (s2 (if (and $word$ (member 'adv $p-o-s))
     go s3
     )
     (if (and $word$ (equal $word '#))
     go success
       after
                  (setq fix (append fix (list (caar num_list))))
                  (setq fixuplist (append (list (list (caar num_list)))
                                 fixuplist) )
                  (setq num_list (cdr num_list))
     )
     (if $ord-nr$
     go s2
     )
     (else go success)
 )%s2
 (s3 (if $ord-nr$
    go success
    )
  )%s3
)%$nr-adj$


(define-atn
       $n-p$ ()
 (s1 (if (and $word$ (member 'det $p-o-s))
    go s2
    )
    (if (and $word$ (equal $word '$))
    go success
    )
    (else go s2)
 )%s1
 (s2 (if $ord-card$
    go s3
    )
    (if (and $word$ (member 'adv $p-o-s))
    go s3
      after
              (setq fix (append fix (list $word)))
    )
    (if $nr-adj$
    go s3
    )
 )%s2
 (s3 (if (and $word$ (member 'noun $p-o-s))
    go s4
      after
        (cond ((and (null (equal $word '#)) (null (equal $spec 'text)) )
              (setq kind $spec)
               )
             (t kind)
        )%cond
    )
    (else go s4)
 )%s3
 (s4 (if (and $word$ (equal $word '#))
    go s4
      after
                  (setq fix (append fix (list (caar num_list))))
                  (setq fixuplist (append (list (list (caar num_list)))
                           fixuplist) )
                  (setq num_list (cdr num_list))
    )
    (if (and $word$ (equal $word 'and))
    go s1
    )
```

```
      (else go success)
  )%s4
 )%$n-p$


(define-atn
      $ord-card$ ()
 (s1 (if (and $word$ (member 'adv $p-o-s))
     go s2
       after
           (setq ordv $word)
     )
 )%s1
 (s2 (if (and $word$ (equal $word '#))
     go success
       after
           (setq fix (append fix (list ordv (caar num_list))))
           (setq fixuplist (append (list (list (caar num_list) ordv))
                                             fixuplist) )
           (setq num_list (cdr num_list))
     )
 )%s2
)%$ord-card$


(define-atn
         $ord-nr$ ()
 (s1 (if (and $word$ (equal $word '#))
     go s2
       after
      (cond (fix
                    (setq fix (append fix (list (caar num_list))))
            )
            (t
               (setq fix (list (caar  num_list)))
            )
         )
                    (setq fixuplist (append (list (list (caar num_list)))
                                          fixuplist) )
     )
 )
 (s2 (if (equal 't (cadar num_list))
     go success
       after
                  (setq num_list (cdr num_list))
     )
 )%s2
)$ord-nr$
```

```
(define-atn
          $preposition$ ()
 (s1 (if (and $word$ (member 'prep $p-o-s))
     go s2
       after
                  (cond ((and (member $word '(before after))
                              (member comm '(m k)) )
                             (setq f1 (append f1 (list '*)))
                             (setq fix nil)
                        )
                        (t nil)
                   )
                   (setq check_later (append check_later (list $word)))
     )
 )
 (s2 (if $n-p$
     go success
       after
                 (setq f1 (append f1 (list (append (list kind) fix))))
     )
 )
)%$preposition$
```

```
(de inter-face ()
%This program performs the dialog with the user.
%After the name of the file the user wants to work with is given,
%the program accepts commands. It gives a message if the command
%could not be executed, i.e. in the simulation of the execution
%access to a non-existing line/paragraph was detected.
%If the user inputs the empty list, the program gives a listing of the
%commands in qed that have to be given in order to do what the user
%wants to do.
%After that the user has the option to work with another file or
%to quit.
%called by -
%calls main, statistics, init


  (setq temp 'temporary)
  (setq sequen nil)
  (setq orig nil)
  (print "Give the name of the file you want to work with.")
  (setq filename (read))
  (while filename
  (setq #newfile (compress (list '# (explode filename) '#)))
  (setq sequen nil)
  (remprop filename 'par)
  (remprop #newfile 'par)
  (remprop temp 'par)
  (print "does your file have a title? If no, type ()")
  (setq title (read))
  (cond (title
           (setq offs 3)
        )
        (t (setq offs 0)
        )
  )
  (init)
  (put #newfile 'curr_line 0)
  (put temp 'curr_line 0)
  (put #newfile 'curr_par 0)
  (put temp 'curr_par 0)
  (print "Type your first command.")
  (print "If you want to quit, type ().")
  (setq command (read))
  (while command
     (setq c (main command))
     (cond (c  (setq sequen (append sequen c))
           )
           (t nil)
     )%cond
     (print "If you want to quit, type ().")
     (print "Otherwise type your next command.")
```

```
      (setq command (read))
   )%while command
   (outp sequen)
   (statistics)
   (print "If you want to work with another file, give its name.")
   (print "Otherwise type ().")
   (setq filename (read))
   )%while filename
)%de


(de init ()
%Performs the dialogue with the user in order to be able to
%initialize the variables properly.
%called by inter-face
%calls initl


   (print "How many paragraphs does your file have?")
   (setq nr (put filename 'nr_of_par (read)))
   (put #newfile 'nr_of_par nr)
   (put temp 'nr_of_par nr)
   (initl 1 (addl nr))
   (remprop 'ch_line 'a)
   (remprop 'ch_line 'i)
   (remprop 'ch_line 'd)
   (remprop 'ch_par 'a)
   (remprop 'ch_par 'i)
   (remprop 'ch_par 'd)
)%de init



(de initl (no limit)
%Performs dialogue with the user to initialize variables.
%called by init
%calls itself recursively


   (cond ((lessp no limit)
            (prinl "How many lines does paragraph ")
            (prinl no)
            (print " have?")
            (put filename 'par (append (get filename 'par) (list
                                 (list no (read)))))
            (initl (addl no) limit)
         )
         (t  (put #newfile 'par (get filename 'par))
             (put temp 'par (get #newfile 'par))
         )
```

```
  )
)%de initl



(de statistics  ()
%Prints the information about the current version of the file
%the user is working with.
%called by  inter-face
%calls stat


  (prinl "File ")
  (prinl filename)
  (prinl " now has ")
  (setq nr (get #newfile 'nr_of_par))
  (prinl nr)
  (cond ((equal nr 1)
           (print " paragraph.")
        )
        (t (print " paragraphs.")
        )
  )
  (terpri)
  (stat  1 (add1 nr))
  (remprop #newfile 'par)
  (remprop temp 'par)
  (remprop filename 'par)
)%de statistics



(de stat (no nr)
%Prints out information of the file after all commands
%have been simulated.
%called by  statistics
%calls itself recursively


  (cond ((lessp no nr)
           (prinl "Paragraph ")
           (prinl no)
           (prinl " now has ")
           (setq p (cadr (assoc no (get #newfile 'par))))
           (prinl p)
           (cond ((equal p 1)
                    (print " line.")
                 )
                 (t  (print " lines.")
                 )
```

```
            )
            (stat (add1 no) nr)
        )
        (t nil)
  )
)%de stat



(de outp (out)
%Performs the output of the qed commands.
%called by inter-face
%calls -
  (print "The sequence of commands you have to use is the following:")
  (print 'qed)
  (prin1 "r " )
  (print filename)
  (while out
     (setq out11 (car out))
     (while out11
        (setq out1 (car out11))
        (while out1
           (cond ((equal out1 "1,$p")
                  (prin1 out1)
                  (setq out1 nil)
                 )
                 ((member (car out1) '(a i))
                  (print (car out1))
                  (print (cadr out1))
                  (prin1 (caddr out1))
                  (setq out1 nil)
                 )
                 (t
                  (prin1 (car out1))
                  (setq out1 (cdr out1))
                 )
        )
        )%wh out1
        (terpri)
        (setq out11 (cdr out11))
   )%wh out11
   (setq out (cdr out))
   )%wh out
  (print 'w)
  (print 'q)
)%de outp



(de calc (par_nr)
%Calculates the line number of the first line in paragraph
%par_nr.
%called by ?
%calls itself recursively


  (cond ((lessp 1 par_nr)
             (plus (cadr (assoc (sub1 par_nr) (get filename 'par)))
                   1 (calc (sub1 par_nr)))
        )
        (t offs
        )
  )
)%de calc



(de ? (x y)
%Returns nil if the command cannot be executed, otherwise the
%line number of line x in paragraph y.
%called by ???
%calls calc, adjust, get-dollar


(prog ()

  (cond ((null (numberp y))
             (cond ((equal y 'last)
                       (setq y (get filename 'nr_of_par))
                   )
                   ((equal y 'first)
                       (setq y 1)
                   )
                   ((equal y 'next)
                       (setq y (add1 (get #newfile 'curr_par)))
                   )
                   ((equal y 'previous)
                       (setq y (sub1 (get #newfile 'curr_par)))
                   )
                   (t nil)
%transforms last, first etc. into numbers (for paragraphs)
             )
        )
        (t nil)
  )%cond null
  (cond ((and (null (numberp x)) (atom x))
             (cond ((equal x 'first)
                       (setq x 1)
                   )
                   ((and (equal x 'last) (equal y 0))
                        (setq x (get-dollar))
                    )
                    ((equal x 'last)
                        (setq x (cadr (assoc y (get filename 'par))))
                    )
                    ((equal x 'next)
                        (setq x (add1 (get #newfile 'curr_line)))
                        (setq y 0)
                    )
                    ((equal x 'previous)
                        (setq x (sub1 (get #newfile 'curr_line)))
                        (setq y 0)
                    )
                    (t nil)
%tansforms last, first etc. into numbers (for lines)
            )%cond equal
       )%null
        (t nil)
  )%cond null
  (cond ((and (lessp y (add1  (get filename 'nr_of_par)))
              (greaterp y 0) )
            (cond ((lessp x (add1 (cadr (assoc y (get filename 'par)))))
                        (setq orig (plus x (calc  y)))
                        (return (adjust orig))
                    )
                    (t
                        (prin1 "Line ") (prin1 x)
                        (prin1 " does not exist in paragraph ")
                        (prin1 y) (print ".")
                        (print " Your command cannot be executed.")
                           (return nil)
                      )
            )%cond x
        )
       ((equal y 0)
           (cond ((atom x)
                (cond ((or (lessp x 1) (greaterp x (get-dollar)))
                           (prin1 "Line ") (prin1 x)
                           (print " does not exist.")
                          (print "The command cannot be executed.")
                          (return nil)
                       )
                      (t
                     (setq orig x)
                     (return (adjust x))
                      )
                )
                  )
                 (t (cond ((or (lessp (car x) 1) (greaterp (car x)
                                                       (get-dollar))
                          (prinl "Line ")(prinl (car x))
                          (print " does not exist.")
                          (print "The command cannot be executed.")
                          (return nil)
                        )
                         ((or (lessp (cadr x) 1) (greaterp (cadr x)
                                                   (get-dollar)))
                          (prinl "Line ")(prinl (cadr x))
                          (print " does not exist.")
                          (print "The command cannot be executed.")
                          (return nil)
                         )
                 (t (setq orig x)
                    (return (append (list (adjust (car x)))
                                    (list (adjust (cadr x)))))
                  )
                 )
                 )
             )
        )
        (t
           (prinl "Paragraph ") (prinl y)
           (print " does not exist.")
           (print "Your command cannot be executed.")
            (return nil)
        )%t
 )%cond and
(print "at the end of ? x and y are:  ")
(print x) (print y)
)%prog
)%de ?



(de adjust (x)
%This function finds out, what the real absolute line number is
%in the simulation.
%It returns nil if the line or paragraph referrred to is already deleted
%adjust is called by ?.
%calls -


  (setq subtract 0)
  (setq addto 0)
  (setq del_line (get 'ch_line 'd))
  (cond ((member x del_line)
            (prinl "Line ") (prinl x)
            (print " has been previously deleted.")
            (print "The command cannot be executed.")
            nil
```

```
          )
          (t (while (and del_line (lessp (car del_line) x))
                (setq subtract (subl subtract))
                (setq del_line (cdr del_line))
             )
             (setq app_line (get 'ch_line 'a))
             (while (and app_line (lessp (car app_line )x))
                (setq addto (addl addto))
                (setq app_line (cdr app_line))
             )
             (setq ins_line (get 'ch_line 'i))
             (while (and ins_line (lessp (car ins_line) (addl x)))
                (setq addto (addl addto))
                (setq ins_line (cdr ins_line))
             )
             (setq line (plus x addto subtract))
           )%t
     )%cond
)%de adjust



(de ?? (para)
%If only paragraphs and not line numbers are referred to this function
%is called.
%It is not enough to know, how many paragraphs have been deleted (for the
%counting of the blank lines), but also which ones (for the absolute
%line number).
%called by extract
%calls sum-up


(prog ()
  (setq par_begin 0)
lab
  (cond ((atom  para)
             (setq par para)
        )
        (t
             (setq par (car para))
        )
  )
  (setq addt 0)
  (setq subtr 0)
  (cond ((null (numberp par))
              (cond ((equal par 'last)
                          (setq par (get filename 'nr_of_par))
                    )
                    ((equal par 'first)
                          (setq par 1)
                    )
                    ((equal par 'next)
                        (setq par (add1 (get #newfile 'curr_par)))
                    )
                    ((equal par 'previous)
                        (setq par (sub1 (get #newfile 'curr_par)))
                    )
                    (t nil)
              )
         )
        (t nil)
   )
   (setq  orig (append orig (list par)))
   (cond ((or (lessp par 1) (greaterp par (get filename 'nr_of_par)))
              (prin1 "Paragraph ") (prin1 par)
              (print " does not exist.")
              (print "The command cannot be executed.")
              (return nil)
         )
         (t nil)
   )
   (setq no_of_lines (sum-up (sub1 par)))
   (setq del_par (get 'ch_par 'd))
   (cond ((member par del_par)
              (prin1 "Paragraph ") (prin1 par)
              (print "has been previously deleted.")
              (print "The command cannot be executed.")
              (return nil)
         )
         (t nil)
   )%cond member
  (while (and del_par (lessp (car del_par) par))
     (setq subtr (sub1 subtr))
     (setq no_of_lines (difference no_of_lines (cadr (assoc
                             (car del_par) (get filename 'par))) ))
     (setq del_par (cdr del_par))
  )%while
   (setq app_par (get 'ch_par 'a))
   (while (and app_par (lessp (car app_par ) par))
      (setq addt (add1 addt))
      (setq no_of_lines (plus no_of_lines (cadr (assoc (car app_par)
                                              (get #newfile 'par)))))
      (setq app_par (cdr app_par))
   )
   (setq ins_par (get 'ch_par 'i))
   (while (and ins_par (lessp (car ins_par) (add1 par)))
      (setq addt (add1 addt))
      (setq no_of_lines (plus no_of_lines (cadr (assoc (car
                                 ins_par ) (get #newfile 'par)))))
      (setq ins_par (cdr ins_par))
  )
  (setq par (plus par addt subtr))
  (cond ((equal (length para) 2)
           (setq par_begin (plus offs no_of_lines (sub1 par) subtr addt))
           (setq para (cadr para))
           (go lab)
        )
           (t (cond ((equal par_begin 0)
                        (setq par_begin (plus offs no_of_lines
                                    (sub1 par) subtr addt))
                        (return (list par_begin (sub1 (plus par_begin
                               (cadr (assoc (plus par subtr addt)
                                          (get #newfile 'par))))))
                    )
                    (t (return (list par_begin
                               (sub1 (plus offs no_of_lines (sub1 par)
                                   subtr addt (assoc (plus par subtr
                                     addt)(get #newfile 'par))))))
                    )
                )
           )
      )
)%prog
)%de



(de sum-up (lim)
%Sums the number of lines for the first lim paragraphs.
%called by ??
%calls itself recursively


  (cond ((lessp 0 lim)
           (plus (sum-up (sub1 lim)) (cadr (assoc lim (get filename
                                                         'par))))
       )
       (t 0)
  )
)%de sum-up



(de merge (a b)
%Merges 2 sorted lists into 1 sorted list
%called by update-l, update-p
%calls itself recursively


  (cond ((null a) b)
        ((null b) a)
        (t (cond ((lessp (car a) (car b))
                     (cons (car a) (merge (cdr a) b))
                 )
                 (t (cons (car b) (merge a (cdr b)))
                 )
           )
        )
  )
)%de merge



(de update-1 (x)
%Updates the property lists of ch_par, ch_line and filen
%one line at a time
%called by comm-distr
%calls merge, beg, adjul, ad-asso


  (setq new nil)
  (put 'ch_line comm (merge (get 'ch_line comm) (list x)))
%find the par, in which the line is
  (setq x1 x)
  (setq para 0)
  (setq x1 (difference x1 offs))
  (while (greaterp x1 0)
     (setq para (add1 para))
     (setq x1 (difference x1 (plus (cadr (assoc para (get #newfile
                                          'par))) 1)))
  )%while
%insert the par-# at the right place of the property com of ch_par
  (setq para1 (cadr (assoc para (get #newfile 'par))))
  (cond ((or (equal comm 'a) (equal comm 'i) )
             (setq para1 (add1 para1))
        )
        (t (setq para1 (sub1 para1)) )
  )
  (cond ((lessp para1 1)
             (put #newfile 'nr_of_par (sub1 (get #newfile 'nr_of_par)))
             (cond ((equal (get #newfile 'nr_of_par) 0)
                    (print "Your file does not contain any lines any more")
                   )
                   (t
                      (put #newfile 'par (append (beg (sub1 para))
                              (adjul par (get #newfile 'nr_of_par))))
                    )t
             )
        )%lessp para1 1
        (t %para1 /= 0
          (ad-asso para paral)
        )%t
   )%cond
)%de



(de update-p (p)
%Updates the properties of #newfile and ch_line.
%called by
%calls merge, beg, adjul, adju2


   (cond ((equal comm 'd)
            (setq low (addl (sum-up (subl p))))
            (setq high  (plus low (cadr (assoc p (get filename 'par)))))
            (setq c nil)
            (while (greaterp high low)
               (setq c (append c (list low)))
               (setq low (addl low))
            )
            (put 'ch_line 'd (menge (merge (get 'ch_line 'd) c)))
            (put #newfile 'par (append (beg (subl p)) (adjul p
                                        (get #newfile 'nr_of_par))))
            (put #newfile 'nr_of_par (subl (get #newfile 'nr_of_par)))
         )
         (t (put #newfile 'par (append (beg p)
                              (adju2 (addl p) (get #newfile
                                                 'nr_of_par))))
           (put #newfile 'nr_of_par (addl (get #newfile 'nr_of_par)))
           (setq l11 (cu-par p))
           (setq l0 0)
           (setq l12 nil)
           (while (lessp l0(cadr (assoc p (get #newfile 'par))))
              (setq l12 (cons l11 l12))
              (setq l0 (addl l0))
           )
           (put 'ch_line comm (merge l12 (get 'ch_line comm)))
         )
)
)%de update-p



(de adjul (lo hi)
Adjusts the association list of the property par of #newfile.
%called by update-l, update-p
%calls itself recursively
```

```
  (cond ((lessp lo hi)
              (append (list (list lo (cadr (assoc (add1 lo)
                                        (get #newfile 'par))))
                     (adju1 (add1 lo) hi))
        )
        (t nil)
  )
)%de adju1



(de adju2 (lo hi)
%Adjusts the association list of the property par of #newfile.
%called by update-p
%calls itself recursively


  (cond ((lessp lo hi)
              (append (adju2 lo (sub1 hi))
              (list (list (add1 hi) (cadr (assoc hi
                                                 (get #newfile 'par))))))
        )
        (t nil)
  )
)%de adju2



(de beg (count)
%Copies the part of the association list before the paragraph
%being worked on.
%called by update-1, update-p
%calls itself recursively


  (cond ((lessp 0 count)
               (append (beg (sub1 count))
               (list (list count (cadr (assoc count
                                              (get #newfile 'par))))))
        )
        (t nil)
  )
)%de beg



(de ad-asso (p1 p)
%Adds a new element to the association list of par of #newfile
%called by update-1
%calls -
```

```
  (setq count1 1)
  (setq res nil)
  (while (lessp count1 p1)
     (setq res (append res (list (assoc count1 (get #newfile 'par)))))
     (setq count1 (add1 count1))
  )
  (setq res (append res (list (list p1 p))))
  (while (lessp count1 (get #newfile 'nr_of_par))
     (setq count1 (add1 count1))
     (setq res (append res (list (assoc count1 (get #newfile 'par)))))
  )
  (put #newfile 'par res)
)%de ad-asso



(de cu-line (x)
%Calculates current paragraph number from current line number
%called by comm-distr
%calls -


  (setq p 0)
  (put #newfile 'curr_line x)
  (while (greaterp x -1)
     (setq p (add1 p))
     (setq x (difference (sub1 x) (cadr (assoc p (get filename 'par)))))
  )%while
  (put #newfile 'curr_par p)
)%de cu-line



(de cu-par (p)
%Calculates the current line number from the current paragraph
%called by comm-distr
%calls -


  (setq x 0)
  (put #newfile 'curr_par p)
  (while (greaterp p 1)
     (setq p (sub1 p))
     (setq x (plus x 1 (cadr (assoc p (get filename 'par)))))
  )
  (put #newfile 'curr_line x)
)%de cu-par
```

```
(de menge (x)
%Transforms an ordered list into an ordered set
%called by ??
%calls itself recursively


  (cond ((null (cdr x))
             x
        )
        ((equal (car x) (cadr x))
            (menge (cons (car x) (cddr x)))
        )
        (t  (cons (car x) (menge (cdr x)))
        )
  )
)%de menge



(de main (sen)
%Initializes all necessary variables.
%called by inter-face
%calls prepare-input
  (setq c_num '(one two three four five six seven eight nine ten
                eleven twelve))%cardinal numbers
  (setq o_num '(first second third fourth fifth sixth seventh
                eighth ninth tenth eleventh twelvth))
  (prepare-input sen)
)%de main



(de prepare-input (sen)%for the atn's
%This program scans an input sentence. If a number is encountered
%trace _num is called.
%The number is replaced by # and is added to num_list.
%Num_list consists of all numbers in the sentence in the original
%sequence and indicators, whether the number is a cardinal number (t)
%or an ordinal number.
%If text in " is encountered, it is put on the text_list and its
%occurrence is marked by a $ in the sentence.
%If an identifier is encountered, build-word is called.
%Punctuation marks in a sentence are ignored and
%eliminated.
%called by main
%calls add-and, parse, comm-distr, trace-num, build-word, split


(prog (inp comm num_list input retl text_list)
  (setq punct_marks '( !' !: !; !? !!) )
```

```
10
  (setq input nil)
  (setq retl nil)
  (setq inp sen)
11
  (cond ((null inp)
              (setq input (add-and input))
              (terpri)(terpri)(terpri)
              (setq f2 nil)
              (cond ((parse input)
                        (setq orig nil)
                        (setq f2 (rem f2 '(nil)))
                        (cond ((member comm '(ma mi ka ki))
                                  (setq seql (moving f2))
                              )
                              ((member 'file input)
                                  (setq seql (list (cons comm
                                                   (list 'file))))
                              )
                              (t
                                  (setq seql (split f2))
                              )
                        )
                        (setq r 't)
                        (while (and seql r)
                           (setq comm (caar seql))
                           (setq r (comm-distr (car seql)))
                           (setq retl (append retl (list r)))
                           (setq seql (cdr seql))
                        )
                        (cond (r
                                 (transf temp filen)
                                 (transfl 'ch_l 'ch_line)
                                 (transfl 'ch_p 'ch_par)
                                 (return (list retl))
                              )
                              (t
                                 (transf filen temp)
                                 (transfl 'ch_line 'ch_l)
                                 (transfl 'ch_par 'ch_p)
                                 (return nil)
                              )
                        )
                    )
                    (t nil)
              )
        )%input is traced till the  end
        ((numberp (car inp))
            (setq num_list (append num_list
                           (trace-num))
            (setq input (append input (list '#)))
            (setq inp (cdr inp))
            (go 11)
        )
        ((equal (car (explode (car inp))) '!")(setq text_list
                     (append text_list (list (car inp))) )
            (setq input (append input (list '$)))
            (setq inp (cdr inp))
            (go 11)
        )
        ((member (car inp) punct_marks) (set inp (cdr inp)) (go 11))
        ((idp (car inp)) (setq input (append input
                                            (list (build-word)))
            (setq inp (cdr inp))
            (go 11)
        )
        (t (print "error") )
    )%cond
)%prog
)%de prepare-input



(de trace-num ()
%If a number is a cardinal number, a list of the number and nil
%is returned.
%If the number is an ordinal number, which is detected by
%checking whether the next word in the sentence is st, nd, rd,
%or th, a list of the number and t is returned.
%called by prepare-input
%calls -


(prog()
  (setq i (car inp))
  (cond ((and (cdr inp) (member (cadr inp) '(st nd rd th)))
            (setq inp (cdr inp))
            (return (list(list i  't)))
        )
        (t (return (list (list i nil)))   )
  )%cond
)%prog
)%de trace-num



(de build-word ()
%If a punctuation mark follows a word without a blank,
%this punctuation marks is deleted.
%If a word is a number-word(1-12), its meaning is found out
```

```
%and a list of the number and t or nil,
%depending on whether the number is an ordinal or
%cardinal number is returned. The word is set to #.
%Otherwise the word is returned unchanged.
%called by prepare-input
%calls -


(prog ()
  (setq word (car inp))
  (cond ((member (car (reverse (explode word))) punct_marks)
          (setq word (compress (reverse (cdr (reverse (explode word
                            )))) )))    )
        %get rid of punct_mark
  (cond ((member word c_num)
          (cond ((equal word 'one) (setq num_list (append num_list
                                     (list (list '1 'nil)))) )
                ((equal word 'two) (setq num_list (append num_list
                                     (list (list '2 'nil)))) )
                ((equal word 'three) (setq num_list (append num_list
                                     (list (list '3 'nil)))) )
                ((equal word 'four) (setq num_list (append num_list
                                     (list (list '3 'nil)))) )
                ((equal word 'five) (setq num_list (append num_list
                                     (list (list '5 'nil)))) )
                ((equal word 'six) (setq num_list (append num_list
                                      (list (list '6 'nil)))) )
                ((equal word 'seven) (setq num_list (append num_list
                                      (list (list '7 'nil)))) )
                ((equal word 'eight) (setq num_list (append num_list
                                      (list (list '8 'nil)))) )
                ((equal word 'nine) (setq num_list (append num_list
                                      (list (list '9 'nil)))) )
                ((equal word 'ten) (setq num_list (append num_list
                                      (list (list '10 'nil)))) )
                ((equal word 'eleven) (setq num_list (append num_list
                                      (list (list '11 'nil)))) )
                ((equal word 'twelve) (setq num_list (append num_list
                                        (list (list '12 'nil))))
                         )
          )%word
          (return  '#)
        )%cond cardinal mumber
        ((member word o_num)
          (cond ((equal word 'first)
                    (cond ((and (numberp (cadr inp))(null(member
                                 (cddar inp) '(st nd rd th))))
                                       (return word))
                          (t (setq num_list (append num_list
                                        (list (list '1 't)))) )
```

```
                        )%cond
                    )%equal
                    ((equal word 'second) (setq num_list (append num_list
                                                     (list (list '2 't)))) )
                    ((equal word 'third) (setq num_list (append num_list
                                                     (list (list '3 't)))) )
                    ((equal word 'fourth) (setq num_list (append num_list
                                                     (list (list '4 't)))) )
                    ((equal word 'fifth) (setq num_list (append num_list
                                                     (list (list '5 't)))) )
                    ((equal word 'sixth) (setq num_list (append num_list
                                                      (list (list '6 't)))) )
                    ((equal word 'seventh) (setq num_list (append num_list
                                                     (list (list '7 't)))) )
                    ((equal word 'eighth) (setq num_list (append num_list
                                                     (list (list '8 't)))) )
                    ((equal word 'ninth) (setq num_list (append num_list
                                                     (list (list '9 't)))))
                    ((equal word 'tenth) (setq num_list (append num_list
                                                     (list (list '10 't)))) )
                    ((equal word 'eleventh)(setq num_list (append num_list
                                                      (list (list '11 't)))) )
                    ((equal word 'twelvth) (setq num_list (append num_list
                                                       (list (list '12 't)))) )
              )
              (return '#)
          )%cond ordinal number
          (t (return word))
  )
)%prog
)%de build-word



(de add-and (sentence)
%After a sentence is scanned, it is checked, whether 2 numbers follow
%each other without an 'and' in between. In this case the 'and' is
%fit in.
%If 2 numbers are connected by 'to', 'thru', or 'through', the
%intermediate values connected with and are added in between.
%The num_list is changed appropriately for both cases.
%called by prepare-input
%calls -


(prog (sent lower_bound upper_bound help nl)
l1
  (cond ((and sentence (cdr sentence))
            (cond ((and (equal (car sentence) '#)
                        (equal (cadr sentence) '#))
                    (setq sent (append sent (list (car sentence)
                                                  'and)))
                    (setq nl (append nl  (list (car num_list))))
                    (setq num_list (cdr num_list))
                    (setq sentence (cdr sentence))
                    (go l1)
                )
                ((and (equal (car sentence) '#)
                      (member (cadr sentence) '(thru through to))
                      (equal (caddr sentence) '#))
%e.g. 7 thru 9 -> 7 and 8 and 9
                    (setq lower_bound (caar num_list))
                    (setq upper_bound (caadr num_list))
                    (cond ((equal lower_bound upper_bound)
                              (setq sent (append sent
                                             (list (car sentence))))
                              (setq num_list (cdr num_list))
                              (setq sentence (cdddr sentence))
                              (go l1)
                          )%equal
                          ((greaterp lower_bound upper_bound)
                             (setq help lower_bound)
                             (setq lower_bound upper_bound)
                             (setq upper_bound help)
                          )%gt
                          (t nil)
                    )%cond
                    (setq num_list (cdr num_list))
                    (while (lessp lower_bound upper_bound)
                            (setq sent (append sent (list '# 'and)))
                            (setq nl (append nl (list
                                      (list lower_bound
                                      (cadar num_list)))))
                            (setq lower_bound (add1 lower_bound))
                    )%while
                    (setq sentence (cddr sentence))
                    (setq num_list (append (list (list upper_bound
                             (cadar num_list)))(cdr num_list)))
                    (go l1)
              )%and or
                (t (setq sent (append sent (list (car sentence))))
                   (cond ((equal (car sentence) '#)
                             (setq nl (append nl (list
                                                  (car num_list))))
                             (setq num_list (cdr num_list))
                         )
                         (t nil)
                  )
                   (setq sentence (cdr sentence))
                    (go l1)
```

```
                 )%t
           )%cond
           (go 11)
         )%and
         (sentence (setq sent (append sent sentence))
                   (cond ((equal (car sentence) '#)
                                 (setq nl (append nl (list (car num_list))))
                          )
                         (t nil)
                   )
                   (setq num_list nl)
                   (return sent)
         )%sentence
         (t (setq num_list nl) (return sent))
   )%cond
)%prog
)%de add-and



(de split (arg)
%Splits the command in a sequence of commands, each referring to
%one location.
%called by prepare-input, moving
%calls match1, match2, match3, x-change


  (setq m2 nil)
  (setq ret nil)
  (cond ((equal (length arg) 1)
            (cond ((and (caar arg) (member (car arg) 'file))
                        (setq ret (list (cons comm (car arg))))
                        (setq mi nil)
                  )
                  (t
                        (setq mi (match1 (car arg)))
                  )
            )
        )
        ((equal (length arg) 2)
            (cond ((and (member 'paragraph (car arg))
                         (null (member 'paragraph
                                                 (cadr arg))))
                        (setq mi (match2 (cadr arg) (car arg) nil))
                  )
                  ((setq mi (match3 (car arg) (cadr arg)))
                        nil
                  )
                  (t  (setq mi (match2 (car arg) (cadr arg) nil))
                  )
```

```
          )
     )
     ((equal (length arg) 4)
         (cond ((and (member 'paragraph (car arg))
                     (null (member 'paragraph
                                     (cadr arg))))
                   (setq mi (match2 (cadr arg) (car arg) nil))
               )
               (t  (setq mi (match2 (car arg) (cadr arg) nil))
               )
         )
         (cond ((and (member 'paragraph (caddr arg))
                     (null (member 'paragraph
                              (cadddr arg))))
                    (setq mi (match2 (caddr arg)
                                (caddr arg) nil))
               )
               (t (setq mi (match2 (caddr arg)
                              (cadddr arg) nil))
               )
         )
     )
     (t
          (cond ((member 'paragraph (car arg))
                     (setq m2 (match2 (cadr arg)
                                 (car arg) nil))
                     (setq mi (match2 (caddr arg)
                                 (car arg) nil))
                )
                (t  (setq m2 (match1 (car arg)))
                    (cond ((member 'paragraph (cadr arg))
                                (setq mi (match2 (caddr arg)
                                            (cadr arg) nil))
                          )
                          (t  (setq mi (match2 (cadr arg)
                                           (caddr arg) nil))
                          )
                    )
                )
     )
 )
)
(cond (m2
         (while m2
            (setq m3 mi)
            (while m3
               (setq ret (append ret (list (list
                     comm (car m2) (car m3)))))
               (setq m3 (cdr m3))
            )%while
            (setq m2 (cdr m2))
         )%while m2
      )
      (t (while mi
            (setq ret (append ret (list
                  (cons comm (car mi)))))
            (setq mi (cdr mi))
         )%wh
      )
  )
)%de split



(de match1 (lp)
%Builds the proper sequence of commands if referring
%to several locations.
%called by split
%calls -


  (setq seq nil)
  (setq l1 (car lp))
  (setq lr (cdr lp))
  (while lr
     (cond ((or (numberp (car lr)) (null (cdr lr)))
                (setq seq (append seq (list
                            (list l1 (car lr)))))
                (setq lr (cdr lr))
           )
           (t  (setq seq (append seq (list (list
                          l1 (car lr) (cadr lr)))))
               (setq lr (cddr lr))
           )
     )
  )%wh
  seq
)%de match1



(de match2 (l p x)
%Builds the proper sequence of commands if referring to
%several distinct locations.
%called by split, match3
%calls -


  (setq l1 (cdr l))
  (setq p2 (cdr p))
```

```
(setq l (car l))
(setq p (car p))
(setq seq nil)
(while l1
   (cond ((or (numberp (car l1))(null (cdr l1)))
              (setq pl p2)
              (while pl
                 (cond ((or (numberp (car pl))
                            (null (cdr pl)))
                          (cond (x
                              (setq seq (append seq (list
                                (list (list l (car l1))
                                (list p (car pl))))))
                                )
                                (t
                              (setq seq (append seq (list
                                (list l (car l1) p (car pl)
                                       ))))
 )%t
 )%cond
                              (setq pl (cdr pl))
                                )
                     (t (cond (x
                              (setq seq (append seq (list
                                (list (list l (car l1))
                                (list p (car pl)
                                        (cadr pl))))))
                              )
                              (t
                              (setq seq (append seq (list
                                  (list l (car l1) p (car pl)
                                    (cadr pl)))))
 )%t
 )%cond
                         (setq pl (cddr pl))
                     )
              )
        )
        (setq l1 (cdr l1))
     )
     (t  (setq pl p2)
         (while pl
            (cond ((or (numberp (car pl)
                       (null (cdr pl)))
                       (cond (x
                                (setq seq (append seq (list
                                  (list (list l
                                     (car l1) (cadr l1))
                                  (list p (car pl))))))
                              )
```

```
                                             (t
                                               (setq seq (append seq (list
                                                 (list l (car ll)
                                                   (cadr ll) p (car pl)
                                                      ))))
                                             )
                                        )
                                        (setq pl (cdr pl))
                                   )
      (t (cond (x
                 (setq seq (append setq (list (list (list
                         l (car ll) (cadr ll))
                     (list p (car pl) (cadr pl))))))
               )
               (t  (setq seq (append seq (list (list l (car ll)
                                 (cadr ll) p (car pl) (cadr pl)))))
               )
         )
         (setq pl (cddr pl))
      )
     )
    )
    (setq ll (cddr ll))
    )%t
  )%cond
  )%wh
  seq
)%de match2



(de match3 (l p)
%Is called if references to lines or paragraphs
%are connected by 'and'.
%called by split
%calls match2


  (cond ((or (and (member 'line l)
                  (member 'line p))
             (and (member 'paragraph l)
                  (member 'paragraph p)) )
                     (match2 l p 't)
        )
        (t nil)
  )
)%de match3
```

```
(de rem (x y)
%Removes item y from the list x
%called by inter-face
%calls itself recursively


  (cond ((null x) x)
        ((equal (car x) y)
            (rem (cdr x) y)
        )
        (t  (cons (car x) (rem (cdr x) y)))
        )
  )
)%de rem



(de comm-distr (qed)
%Calls different functions depending on the command and
%then assembles the command in a syntactically correct form.
%called by prepare-input
%calls extract, update-l, cu-line, update-p


(prog (what where)
  (setq ql (car qed))
  (cond ((member 'file qed)
            (cond ((equal ql 's)
                      (setq ret nil)
                      (while text_list
                         (setq ret (append ret (list "1,$s/"
                           (car text_list) "/" (cadr text_list)
                                            "/")))
                         (setq text_list (cddr text_list))
                      )%while
                      (put filen 'curr_line (get-dollar))
                      (put filen 'curr_par (get filen 'nr_of_par))
                      (return ret)
                  )
                  ((equal ql 'p)
                      (put filen 'curr_par (get filen 'nr_of_par))
                      (put filen 'curr_line (get-dollar))
                      (return "1,$p")
                  )
                  (t nil)
            )
        )%cond member
        (t  nil)
  )
  (cond ((equal ql 'mi)
    (setq where (extract (caddr qed)))
    (cond ((null where)
                (return nil)
          )
          ((atom where)
               where
          )
          (t (setq where (car where))
          )
    )
    (setq qed (cons 'i (cadr qed)))
    (setq ql "zm0")
    (setq cl 'd)
    (setq c2 'i)
)
((equal ql 'ma)
    (setq where (extract (caddr qed)))
    (cond ((null where)
                (return nil)
          )
          ((atom where)
               where
          )
          (t (setq where (cadr where))
          )
    )
    (setq qed (cons 'a (cadr qed)))
    (setq ql "zm0")
    (setq c2 'a)
    (setq cl 'd)
)
((equal ql 'ki)
    (setq where (extract (caddr qed)))
    (cond ((null where)
                (return nil)
          )
          ((atom where)
                 (setq where (subl where))
          )
          (t (setq where (car where))
          )
    )
    (setq qed (cons 'i (cadr qed)))
    (setq ql "zk0")
    (setq cl nil)
    (setq c2 'i)
)
((equal ql 'ka)
    (setq where (extract (caddr qed)))
    (cond ((null where)
                    (return nil)
                )
                ((atom where)
                     where
                )
                (t (setq where (cadr where))
                )
          )
          (setq qed (cons 'a (cadr qed)))
          (setq q1 "zk0")
          (setq c1 nil)
          (setq c2 'a)
      )
      (t nil)
)
(setq what (extract (cdr qed)))
(cond ((null what)
          (return nil)
      )
      ((atom what)
          (cond ((equal q1 's)
                    (setq ret (list what q1 "/" (car text_list)
                          "/" (cadr text_list) "/p"))
                    (setq text_list (cddr text_list))
                    (cu-line what)
                )
                ((or (equal q1 'i)(equal q1 'a))
                    (setq ret (list what q1 (car text_list) "\f"))
                    (setq text_list (cdr text_list))
                    (update-1 what)
                    (cu-line what)
                )
                (t (cond (where
                            (setq ret (list where "p" what q1))
                            (cond (c1
                                      (setq comm 'd)
                                      (update-1 what)
                                      (setq comm c2)
                                      (update-1 where)
                                  )
                                  (t  (setq comm c2)
                                      (update-1 where)
                                  )
                            )
                            (cu-line where)
                         )
                         (t  (setq ret (list what q1))
                             (update-1  orig)
                             (cu-line  orig)
                         )
```

```
              )
           )
        )
    )
    ((equal ql 'i)
        (setq ret (list (car what) ql
               (car text_list)  "\f") )
        (setq text_list (cdr text_list))
        (update-l (car what))
        (cu-line (car what))
    )
    ((equal ql 'a)
        (setq ret (list (cadr what) ql
               (car text_list)  "\f") )
        (setq text_list (cdr text_list))
        (update-l (cadr what))
        (cu-line (cadr what))
     )
     ((equal ql 's)
         (setq ret (list (car what) "," (cadr what) ql "/"
               (car text_list) "/" (cadr text_list) "/"))
         (setq text_list (cddr text_list))
         (cu-line (cadr what))
    )
    (t  (cond (where
                 (setq ret  (list  where "p" (car what) ","
                                   (cadr what) ql))
                 (cond (cl
                          (setq comm cl)
                          (cu-line (car what))
                          (update-p (get filen 'curr_par))
                          (setq low (car what))
                          (setq high (cadr what))
                          (while (lessp low (addl high))
                             (update-l low)
                             (setq low (addl low))
                          )
                          (setq comm c2)
                          (cu-line where)
                          (update-p (get filen 'curr_par))
                          (setq low (car what))
                          (setq high (cadr what))
                          (while (lessp low (addl high))
                             (update-l where)
                             (setq low (addl low))
                          )
                       )
                       (t %cl = nil
                          (setq comm c2)
                          (cu-line where)
```

```
                              (update-p (get filen 'curr_par))
                              (setq low (car what))
                              (setq high (cadr what))
                              (while (lessp low (add1 high))
                                 (update-l where)
                                 (setq low (add1 low))
                              )
                       )
                         )
                      )
                      (t  (setq ret (list (car what)     (cadr what) ql))
                          (cu-line (cadr what))
                          (update-p (get filen 'curr_par))
                      )
                 )
              )
   )
   (return ret)
)%prog
)%de comm-distr



(de extract (q)
%Extracts the line and paragraph numbers from the different
%formats of input.
%is called by comm-distr
%calls ?, ??, upd, buildllist, buildplist, ???


   (setq l nil)
   (setq p nil)
   (setq orig nil)
   (cond ((equal (length q) 2)
               (cond ((equal (car q) 'line)
                          (setq l (?  (cadr q) 0))
                     )
                     (t (setq p (?? (cadr q)))
                     )
               )
         )
         ((equal (length q) 3)
               (cond ((equal (car q) 'line)
                          (setq l (? (buildllist (cdr q)) 0))
                     )
                     (t (setq pp (buildplist (cdr q)))
                        (setq p (?? pp))
                     )
               )
         )
```

```
        ((equal (length q) 4)
            (setq l (? (cadr q) (car (reverse q))))
        )
        ((equal (length q) 5)
            (cond ((equal (cadr (reverse q)) 'paragraph)
                      (setq l (? (buildllist (list (cadr q) (caddr q)))
                             (car (reverse q))) )
                  )
                  (t (setq p (??? (cadr q) (buildplist
                                  (list (cadr (reverse q))
                                       (car (reverse q))))))
                  )
            )
        )
        (t  (setq q1 (buildllist (list (cadr q) (caddr q))))
            (setq q2 (buildplist (list (cadr (reverse q))
                                      (car (reverse q)))))
            (setq l (??? q1 q2))
        )
  )
  (cond (l l)
        (p p)
        (t nil)
  )
%to return the correct values
)%de extract



(de upd (l)
%l is a par-# w.r.t the original file
%upd returns the par-# w.r.t. the current file
%called by extract
%calls -


  (setq h 0)
  (setq help (get 'ch_par 'd))
  (while (and help (lessp (car help) l))
     (setq h (sub1 h))
     (setq help (cdr help))
  )
  (setq help (get 'ch_par 'a))
  (while (and help (lessp (car help) l))
     (setq h (add1 h))
     (setq help (cdr help))
  )
  (setq help (get 'ch_par 'i))
  (while (and help (lessp (car help) (add1 l)))
     (setq h (add1 h))
     (setq help (cdr help))
  )
  (plus 1 h)
)%de upd



(de ??? (ln ph)
%For references to more than one paragraph this function
%calls ? repeatedly.
%called by extract
%calls ?


(prog ()
  (setq s nil)
  (setq resl nil)
labl
  (cond ((lessp (car ph) (addl (cadr ph)))
             (setq s (? ln (car ph)))
             (cond (s
                        (setq resl (append resl s))
                        (set (car ph) (addl (car ph)))
                        (go labl)
                   )
                   (t  (return nil)
                   )
             )
        )
        (t (return resl)
        )
  )
)%prog
)%de ???



(de buildplist (listl)
%For refernces to more than one paragraph this function builds
%the appropriate list of paragraph numbers.
%called by extract
%calls -


  (setq adv (car listl))
  (setq num (cadr listl))
  (cond ((equal adv 'first)
             (list 1 num)
        )
        ((equal adv 'last)
            (list (add1 (difference (get filename 'nr_of_par) num))
                  (get filename 'nr_of_par) )
        )
        ((equal adv 'previous)
            (list (difference (get filen 'curr_par)  num)
                  (sub1 (get filen 'curr_par)))
        )
         (t (list (add1 (get filen 'curr_par))
                  (plus (get filen 'curr_par) num))
         )
  )
)%de buildplist



(de buildllist (listl)
%For references to more than one line this function returns
%the appropriate list of line numbers.
%called by extract
%calls get-dollar


  (setq adv (car listl))
  (setq num (cadr listl))
  (cond ((equal adv 'first)
            (list 1 num)
        )
        ((equal adv 'last)
            (setq x (get-dollar))
            (list (add1 (difference x num)) x)
        )
        ((equal adv 'previous)
            (list (difference (get filen 'curr_line)  num)
                  (sub1 (get filen 'curr_line)))
        )
        (t  (list (add1 (get filen 'curr_line))
                  (plus (get filen 'curr_line) num))
        )
  )
)%de buildllist



(de get-dollar ()
%Gets the last line in the file.
%called by ?, buildllist
%calls -


  (setq res 0)
  (setq cnt 0)
  (while (lessp cnt (get filename 'nr_of_par))
     (setq cnt (add1 cnt))
     (setq res (plus res (cadr (assoc cnt (get filename 'par)))))
  )
  (plus res (sub1 cnt) offs)
)%de get-dollar
```

APPENDIX B

The following is a sample session of the interface program. The user intends to work on four files with the names docu1, docu2, docu3 and docu4.
This demonstration gives a selection of the variety of commands possible.
Commands that refer to lines are also accepted, if the same format is used referring to paragraphs instead of lines and vice versa.

```
>>START JOURNALIZING
>(inter-face)
"give the name of the file you want to work with."
docu1
"does your file have a title? if no, type ()"
y
"how many paragraphs does your file have?"
3
"how many lines does paragraph "1" have?"
5
"how many lines does paragraph "2" have?"
3
"how many lines does paragraph "3" have?"
8
"type your first command."
"if you want to quit, type ()."
(in my file substitute " it " by " they " and " we " by " you ")
Collecting



Collecting
Collecting
Collecting



((in my file substitute $ by $ and $ by $) accepted)

"if you want to quit, type ()."
"otherwise type your next command."
(also replace " " by "  " in paragraph two)



Collecting
Collecting
```

```
((also replace $ by $ in paragraph #) accepted)

Collecting
"if you want to quit, type ()."
"otherwise type your next command."
(furthermore exchange line 5 and 7)



Collecting



((furthermore exchange line # and #) accepted)

Collecting
Collecting
"if you want to quit, type ()."
"otherwise type your next command."
(would you remove line 2 in para 1)



Collecting
Collecting



((would you remove line # in para #) accepted)

Collecting
"line "5" has been previously deleted."
"the command cannot be executed."
"if you want to quit, type ()."
"otherwise type your next command."
(please display my file)



Collecting



((please display my file) accepted)

"if you want to quit, type ()."
"otherwise type your next command."
()
"the sequence of commands you have to use is the following:"
qed
```

```
"r "docul
"1,$s/"" it ""/"" they ""/""1,$s/"" we ""/"" you ""/"
9","11s"/"" ""/"Collecting
"  ""/"
7"p"5"zk0"
5"p"7"zk0"
5d
7d
"1,$p"
w
q
"file "docul" now has "3" paragraphs."

"paragraph "1" now has "5" lines."
"paragraph "2" now has "3" lines."
"paragraph "3" now has "8" lines."
"if you want to work with another file, give its name."
"otherwise type ()."
docu2
"does your file have a title? if no, type ()"
()
"how many paragraphs does your file have?"
3
"how many lines does paragraph "1" have?"
2
"how many lines does paragraph "2" have?"
4
"how many lines does paragraph "3" have?"
5
"type your first command."
"if you want to quit, type ()."
(delete the last paragraph)



Collecting




((delete the last paragraph) accepted)

Collecting
"if you want to quit, type ()."
"otherwise type your next command."
(add "the" after the 2nd line)




Collecting
Collecting
```

```
Collecting



((add $ after the # line) accepted)

"if you want to quit, type ()."
"otherwise type your next command."
(delete line 2 in para 3)



Collecting
Collecting



((delete line # in para #) accepted)

"line "10" has been previously deleted."
"the command cannot be executed."
"if you want to quit, type ()."
"otherwise type your next command."
(put "this" before the previous line)



Collecting



((put $ before the previous line) accepted)

"if you want to quit, type ()."
"otherwise type your next command."
(insert "end" after the last line)
Collecting



Collecting



((insert $ after the last line) accepted)

Collecting
"if you want to quit, type ()."
"otherwise type your next command."
(move line 5 after line 7)
```

```
Collecting
Collecting



((move line # after line #) accepted)

"line "7" has been previously deleted."
"the command cannot be executed."
"if you want to quit, type ()."
"otherwise type your next command."
()
"the sequence of commands you have to use is the following:"
qed
"r "docu2
8","12d
2a
"the"
"\f"
1i
"this"
"\f"
10a
"end"
"\f"
w
q
"file "docu2" now has "2" paragraphs."

"paragraph "1" now has "4" lines."
"paragraph "2" now has "5" lines."
"if you want to work with another file, give its name."
"otherwise type ()."
docu3
"does your file have a title? if no, type ()"
()
"how many paragraphs does your file have?"
3
"how many lines does paragraph "1" have?"
10
"how many lines does paragraph "2" have?"
6
"how many lines does paragraph "3" have?"
7
"type your first command."
"if you want to quit, type ()."
(please remove line 3 from paragraph 2)
```

```
Collecting
Collecting



((please remove line # from paragraph #) accepted)

Collecting
"if you want to quit, type ()."
"otherwise type your next command."
(copy line 3 before line 7)
Collecting



Collecting



((copy line # before line #) accepted)

Collecting
"if you want to quit, type ()."
"otherwise type your next command."
(in the second paragraph would you please erase line 3)
Collecting



Collecting
Collecting



((in the # paragraph would you please erase line #) accepted)

"line "14" has been previously deleted."
"the command cannot be executed."
"if you want to quit, type ()."
"otherwise type your next command."
(in para 4 please delete the first line)
Collecting



Collecting



((in para # please delete the # line) accepted)
```

```
Collecting
"paragraph "4" does not exist."
"your command cannot be executed."
"if you want to quit, type ()."
"otherwise type your next command."
(put "end" after line 30)



Collecting



((put $ after line #) accepted)

"line "30" does not exist."
"the command cannot be executed."
Collecting
"if you want to quit, type ()."
"otherwise type your next command."
(show the file)






((show the file) accepted)

Collecting
"if you want to quit, type ()."
"otherwise type your next command."
()
"the sequence of commands you have to use is the following:"
qed
"r "docu3
14Collecting
d
6"p"3"zk0"
"1,$p"
w
q
"file "docu3" now has "3" paragraphs."

"paragraph "1" now has "11" lines."
"paragraph "2" now has "5" lines."
"paragraph "3" now has "7" lines."
"if you want to work with another file, give its name."
"otherwise type ()."
docu4
```

```
"does your file have a title? if no, type ()"
()
"how many paragraphs does your file have?"
5
"how many lines does paragraph "1" have?"
3
"how many lines does paragraph "2" have?"
7
"how many lines does paragraph "3" have?"
5
"how many lines does paragraph "4" have?"
9
"how many lines does paragraph "5" have?"
4
Collecting
"type your first command."
"if you want to quit, type ()."
(delete line 4 in para 3 and 4)



Collecting
Collecting
Collecting



((delete line # in para # and #) accepted)

Collecting
"if you want to quit, type ()."
"otherwise type your next command."
(delete the next 3 lines)



Collecting



((delete the next # lines) accepted)

Collecting
"if you want to quit, type ()."
"otherwise type your next command."
(delete the previous para)
Collecting
```

```
((delete the previous para) accepted)

Collecting
"if you want to quit, type ()."
"otherwise type your next command."
(delete the first 3 lines)
Collecting



Collecting
```

Since the first paragraph consists of 3 lines, this command has the same effect as the command "delete the first paragraph".

```
((delete the first # lines) accepted)

"if you want to quit, type ()."
"otherwise type your next command."
(delete line 4 and 3 in para 2)



Collecting
Collecting



((delete line # and # in para #) accepted)

"line "16" has been previously deleted."
"the command cannot be executed."
"if you want to quit, type ()."
"otherwise type your next command."
()
"the sequence of commands you have to use is the following:"
qed
"r "docu4
16d
21d
21","23d
12","15d
1","3d
w
q
"file "docu4" now has "2" paragraphs."

"paragraph "1" now has "7" lines."
Collecting
```

```
"paragraph "2" now has "4" lines."
"if you want to work with another file, give its name."
"otherwise type ()."
()
nil
>!jour off
>>STOP JOURNALIZING
```