THE DESIGN AND IMPLEMENTATION OF PRONTO PROCESSOR FOR NATURAL TEXT ORGANIZATION

by

Steven Michael Anderson

B.S., Kansas State University, 1981

A MASTERS' REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1983

Approved by:

Ving Wollenteni
Major Professor

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

The purpose of this report is to describe the design of a programming language, using natural language as its base. This language is aimed at providing a tool for knowledge representation.

Knowledge engineering is a growing discipline in the area of artificial intelligence (AI). Much of the current focus in knowledge representation is in the development of high level languages to aid in providing structures for "patterns of inference." These knowledge representation languages are usually supported by embedding in host languages. The most common of these host languages is LISP, which is a powerful symbol manipulation language capable of supporting knowledge representation. Although LISP supplies a powerful modeling tool, it is not the first phase in designing knowledge descriptions. Most descriptions are formulated using natural language; thus the purpose of this new language is to develop a friendly user tool for natural knowledge descriptions.

Several approaches to natural language processing have been attempted over the years. Much of the focus of knowledge engineering has been aimed at developing intelligent systems. This area of knowledge engineering is extremely useful and will prevail as such, but the present state of the art has focussed much of its efforts on present computer languages and tools. These present tools have sufficed in offering a

means for studying human and computer interfaces in areas such as knowledge engineering, but growing trends indicate needs for more sophisticated tools. Natural language would appear to be ideally suited to the task, as it is a natural first step in the development of knowledge descriptions. The goal then is to develop a sophisticated natural language processor (NLP) to meet the growing needs for better knowledge engineering tools.

There are several computer languages and language tools presently available to use in developing NLP's. A very modest, but practical approach to use would be to choose an embedded language well suited to handling natural language text. Current machine architectures support this idea, as is seen through many word processing machines. Well written word processors support almost all of the basic data operations needed for such an NLP to be built. Additions will need to be added to make such a new language a proper virtual machine processor. However, a readily available base machine has been established.

This new language for AI based on a sophisticated word processing virtual machine is called PRONTO, or Processor for Natural Text Organization. PRONTO is a symbol manipulation language much like LISP, which incorporates both uniformity of program and data. Like LISP, PRONTO features high user accessibility in both program and structure. These structural features can be embedded in LISP; for example, MACLISP and INTERSLISP supply these capabilities. The goal of PRONTO, however, is to provide this structural feature as a basic attribute.

PRONTO provides the same structure for both program and data, like a word processor, where all text can be easily made visible using

standard display commands. The command sequence is simply part of the text; thus, there is only the need in defining a text interpreter which can handle these command sequences.

The development cycle of most AI interpreters can easily be implemented in most languages such as LISP or SNOBOL. The development cycle usually takes the form:

1. Host Language
2. Editor
3. Debugger

INTERLISP features this type of structure. Why PRONTO? Syntax is a major drawback in using LISP. Long, deeply nested S-expressions provide for difficult reading, even for well oriented LISP users. Another major drawback to LISP is its difficulty in arriving at user friendly and easily accessible data structures. PRONTO, then, is designed specifically with this in mind.

CHAPTER II

TEXT ORGANIZATION

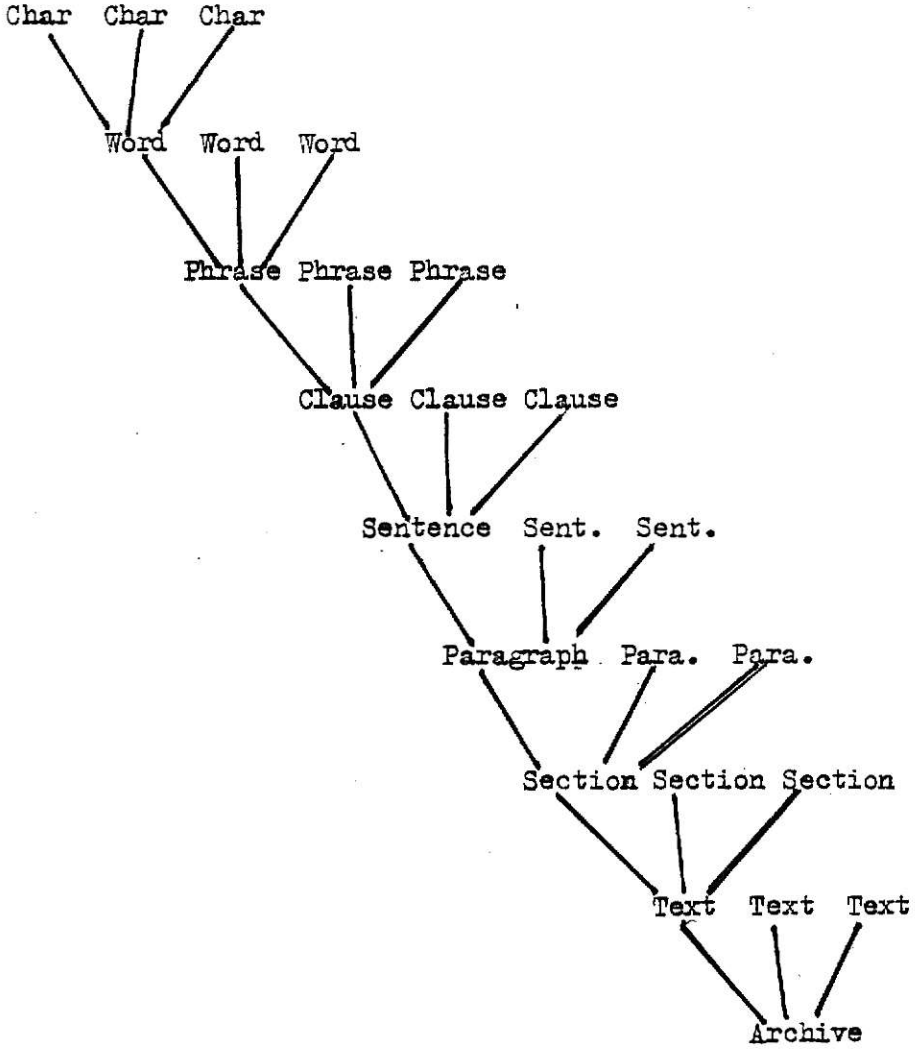
A. TEXT STRUCTURE

Most attempts at natural language processing are aimed at analyzing textual content. The goal here is not to survey textual content, but to understand the content's underlying structure. Knowledge Engineering relies heavily on how well the data is presented both in content and structure. The structure supplies emphasis to the content which enables the structure to describe the casual flow. PRONTO'S concern is defining the structure syntactically in hopes of understanding the underlying description.

Natural language provides its own hierarchy of categories. The levels are, at the most primitive, a character, word, phrase, sentence, paragraph, section, chapter, part, volume, book, series, subject, library. It is obvious that many more levels may be added. It is also very clear that some of the hierarchy may not be necessary or even desirable. (fig. 1) Numeric types could easily be introduced in normal fashion.

B. TEXT DELIMITERS

In analyzing this structure, it must also be noted that each category in the hierarchy has a unique separator associated with it. Some of these separators are not readily apparent, but are very much evident when associated with content or textual layout. Several separators are



This nine level structure may not be adequate (LISP has ability for indefinite number of levels). New levels may be introduced by adding more syntactic conventions (LISP uses brackets for determining levels). This is what we do in Natural Language Text anyway.

fig 1. DATA TYPES

clear in their usage, such as a period or comma and a blank, which depict sentences or phrases respectively. Conceptually, it would be easy to define mnemonic separators for those separators not readily apparent. The category abbreviations:

Character	Eoch (A null -- dummy separator)
Word	Blank
Phrase, Clause	Comma, Colon, Semi-Colon
Sentence	Period, Exclamation, Question Mark
Paragraph	Eopara
Section	Eosect
And so on.	

Needless redundancy is avoided by not inserting lower level separators where higher level separators exist. A notable exception, however, is a period at the end of a paragraph, section, chapter or higher. This might be explained by saying an "eopara" is equal to a "period" plus some formal layout specific notion.

C. USED/MENTIONED TEXT

The hierarchy is incomplete without mention of the need for a reference category. The need for a category to indicate named items or referenced objects is needed in all languages, especially computer languages. There are several ways these reference objects may appear in text; for instance boldface notation or italics provide different types of print. Underlining and capitalization provide another means to highlight or reference pieces of text. This use/mention type of distinction is probably best explained by showing how computer languages use referencing. Variable declarations and keywords are prime examples (DCL VAR__) or (if-then-else). Natural language may be parallelled to this by showing how the title of a book names a book's contents or how a paragraph title names the section of text following the header.

Text can and does contain partial descriptions of the text it is contained within. Many of these partial textual descriptions are based highly on content. This supplies another use/mention distinction of a different kind. For instance, the use of two or more naming conventions to both describe and then to detail what was previously described.

D. TEXT PROCESSING

Text is naturally processed by reading left to right and front to back in a linear type fashion (at least in English). References within the text can detour readers to pictures or other such reference points, but to return to the flow and time passage of the text, the reader must return to the point of departure. This processing parallels the action undertaken in typical computer languages when subprograms or subroutines are involked. The subprogram is called and a detour to the routine is taken. When processing finishes, the subprogram returns to the place from where it was called.

Natural text fails to process procedural definitions without sufficient structural support. Text procedures incorporate the use of naming (numbering) conventions to handle this aspect, such as a numbered sequence of instructions. Computer languages, however, are well suited to procedure handling. PRONTO is not solely natural text, but a product of the structure that underlies the text.

EXAMPLE 1. Use/Mention Self Referenceing Text & Partial Text Description

"This is the first sentence of the second paragraph."

The semantic context contains references (uses) of TEXT STRUCTURES. These references, are themselves, contained within a TEXT STRUCTURE. First sentence refers to the self-contained structure. Second paragraph refers to a higher order text level, further describing the "Text Structure."

EXAMPLE 2. Self referencing Text

"This sentence contains five words."

Sentence refers to the structure; five words refers to the same structure. The text type and the text type's contents are both USED in describing the structure.

EXAMPLE 3. Natural Text Procedural Description

What to do when you get a flat tire on your car.

The first thing to do when encountering a flat tire is to locate the spare tire. Once the spare tire has been located, usually it is in the car's trunk, remove the spare tire and locate it near the flat tire. The next step is to find the jack and all of its parts. Among the jack's parts is usually a lug nut remover/tightener. Remove first the hubcap and then begin unscrewing the lug nuts. After all lug nuts have been loosened a little, insert the jack under the bumper closest to the flat tire, etc. etc.

It should be easy to see that the description above is content driven. Unless PRONTO specifically describes keywords or phrases as syntax procedural, processing using Natural Text structure will be impossible. PRONTO uses text structure to support procedural processing.

EXAMPLE 4. Structure by Sequence of Events

1. Find the spare tire
2. Remove spare tire and place near flat tire
3. Find the jack and its parts (*assemble the jack*)
4. Loosen the lug nuts

.
. etc.

The numbers indicate the specific steps to follow, each in sequence by structure, for PRONTO these numbers would be NAMED text executed by primitives.

CHAPTER III

PRONTO DESIGN PRIMITIVES

A. BASIC CLASSES AND FUNCTIONS

PRONTO is designed around the concepts of word processing architectures. Word processor operations are similar to most other conventional data operations. Conventional systems manipulate bits, bytes and words with operations such as add, shift and move. Word processors manipulate characters, phrases, and sentences up to text with operations such as move, copy and display. Both word processors and conventional languages support a finite set of basic primitives. PRONTO, to manipulate text, must also support this finite set of data operations contained in word processor structures. The goal of PRONTO is to incorporate as many characteristics of natural language text in its design, without degrading the ability to process descriptions procedurally.

A basic set of text handling primitives consists of:

1. move
2. copy
3. find
4. insert
5. delete
6. display

For these text operations to be complete, the concept of a cursor has been added showing location and flow of processing.

PRONTO must also be able to handle descriptions processing using conventional methods. A subroutine call is added to handle procedural processing. PRONTO also uses a second cursor to handle sequence control.

This cursor, a program cursor, serves a similar function as a program counter in conventional systems.

To round out the language's primitive functions, a naming convention is added. This naming convention serves the purpose of allowing PRONTO to have the equivalent of an assignment statement.

This complete set of primitive functions make up the PRONTO language. The data types these functions perform their work on consist of text and text class delimeters (fig. 2). A PRONTO program consists of a sequence of PRONTO primitives strung together to produce a desired behaviour.

B. SYNTACTIC CONVENTIONS

PRONTO not only uses the text classes and text class delimiters seen in fig. 2, but incorporates several other features defined to aid in seeing how the primitive functions will do their work. PRONTO names are handled much the same way that LISP variables are handled -- by direct binding of identifier to value. This means that the text after the name is the value of the variable. Variable bindings become explicit in both name and value. Structure within text is provided by punctuation marks and every data type is terminated by one of a few terminators directly related to it or by any terminator associated with a data type higher in the hierarchy (fig. 1). Each piece of text can then be described as the portion of text between relevant separators, much like LISP'S usage of parenthesis.

Delimited parameters are used in function calls and subprogram calls. The convention employed is to double quote named text and text class descriptors. Single quotes are used within subprogram titles to

ILLEGIBLE DOCUMENT

THE FOLLOWING DOCUMENT(S) IS OF POOR LEGIBILITY IN THE ORIGINAL

THIS IS THE BEST COPY AVAILABLE

TEXT CLASS	TEXT CLASS DELIMITER	ACTUAL IMPLEMENTATION USAGE	DEFAULT VALUES
WORD	SPACE or BLANK	SPACE	SPACE
PHRASE CLAUSE	COMMA COLON or SEMI-COLON	COMMA : or ;	COMMA ;
SENTENCE	PERIOD EXCLAMATION QUESTION	. or ! or ?	PERIOD .
PARAGRAPH	PERIOD & eopara	eopara #	PERIOD .
SECTION	PERIOD & eosect	eosect $	PERIOD .
PART	PERIOD & eopart	eopart %	PERIOD .
BOOK or FILE	End of Text	EOT /	EOT /

fig. 2 PRONTO Text Classes

The USE/MENTION feature employs an "@" symbol to delimit NAMED text.

Each text class is both preceeded and succeeded by its class delimiter or a delimiter of a higher class. Named text delimiters are not in the hierarchy, therefore do not allow replacement or substitution of separators.

delimit parameters. Comments are handled by the use of angle brackets. Comments are ignored in normal processing requirements.

Text descriptors are defined as:

(eow, blank)	end of word
(eoph, eocl)	end of phrase, end of clause
(eos)	end of sentence
(eop)	end of paragraphy
(eosect)	end of section
(eoch)	end of chapter
(eopart)	end of part
(eob)	end of book
(eot)	end of text

When used, text descriptors are quoted.

C. PRONTO RESTRICTIONS

PRONTO is currently being implemented as an interpreter written in LISP. Several programming conventions have been adopted to deal with current LISP implementation and computer facilities available. All text delimiters are explicitly portrayed as non alphanumeric printable characters. Numbers are not part of the current PRONTO hierarchy as LISP treats numbers in a separate way. Single character manipulation is not implemented under the current LISP-PRONTO environment. Storage management is left to the host language. Full screen diagnostics are not implemented in this version of PRONTO, although the design supports the need for full screen capabilities.

CHAPTER IV

PRONTO FUNCTIONS

A. FIND

Finding a pattern in the context of a PRONTO program produces a Boolean result of success or failure. Sequence control is a primary function of the find primitive. In conjunction with find is usually a move command. The program cursor is usually the item moved. A special function (PCM) moves the program cursor to the desired location pending the desired success or failure of the find search.

EXAMPLE 1:

Given the sentence sequence:

```
. The hat is blue.  Find shoes, PCM para back.
Current *PC*
```

And the program segment:

```
find "hat",
PCM "sentence" success
```

Results in:

```
. The hat is blue.  Find shoes, PCM para back.
                    *PC*
```

Transfer of control, where Find shoes, becomes the next instruction.

In the context of the PRONTO System, the find primitive is responsible for locating and distinguishing between named text, subroutines and character strings of varied length. In addition to finding these PRONTO text structures, a specialized command will find the current scope under which the command is being executed. This is simply the current named text hierarchical class.

The find primitive is the primary function involved in controlling the sequence of execution that underlies the PRONTO machine. The find routine establishes the conventions used in placement of the cursors. The find command will always find the previous item to which the find parameter suggests. This convention is adopted for purposes of scoping, error checking only need be done when the find wishes the first item within a scope.

The find command searches forward within the scope of the command from the initial text cursor location. If the match is not made by the end of the scope, find searches from the top of the scope back to its initial search start. The result of the find is success or failure except when finding the scope. On finding the scope the text cursor is reset to the beginning of the current scope and processing proceeds from there.

Find takes one parameter, be it a string or word, name or subroutine. The parameter of the find command may be quoted or unquoted. Quoted parameters are usually processed quicker because the quote acts as a delimiter.

The find command may be executed from the editor at execution level or from a subprogram. The find scope command is executable only at the

execution level.

B. DISPLAY (Text; Class or Structure)

The PRONTO display routine invokes the find primitive to locate the display parameter list for the desired match. If the match is made, the display primitive prints the structure from where the match was made, otherwise the parameter list is displayed.

EXAMPLE 1: Sentence: THIS is a sentence.

Result: Display "THIS"

THIS is a sentence.

EXAMPLE 2: Sentences: (eop) PARAGRAPH with a sentence.

this is the second one (eop)

Display "Paragraph"

Result: PARAGRAPH with a sentence.

This is the second one.

The display function supplies any lower order text class delimiters that are not stored internally as is the case in Example 2, where a period is placed as the final character. Indentation is handled when paragraph delimiters are encountered. Capitalization occurs for the first character (non-blank) following an end of sentence delimiter.

The display function, like the find function, may receive its parameters quoted or unquoted. The display always returns a result no matter whether a parameter has been matched or not. Unmatched parameters return

as entered. This function has entry points at the subprogram level and the execution level.

C. Copy

The PRONTO copy function invokes the find primitive to search for the text to be copied. A temporary copy buffer is issued upon successfully finding the text. Failure results in an appropriate message. The copy is placed directly following the text cursor making the copy of text the next text cursor. Failure to copy leaves the text cursor where it was. Copy uses list surgery when copying text to form a new permanent text structure.

Copy is accessible at both the subprogram level and the execution level. Quoted and unquoted parameters are accepted. The temporary copy buffer is left for the LISP garbage collector. All other global registers are preserved while copy takes place.

D. MOVE

At this stage in PRONTO design, move is not implemented. Move is just a copy that uses two list surgeries to move blocks of text from one section or subprogram into another section or subprogram. Move is destructive to the original text buffer, but builds a new text buffer keeping all pointers untouched by the move intact.

E. INSERT - ADD

PRONTO add is used only at the execution level. Its purpose is to reinitialize input programming. All adds are done to the end of the text buffer. To make a change, with add, a move or copy function must follow.

with the scope of PRONTO, in supplying only primitives to accomplish all user functions.

F. DELETE

Delete is not currently implemented under PRONTO. Delete is just a move to a null area where the garbage collector will take over. Delete will be accessible at both the subprogram level and the execution level. List surgery is involved; changes to original text are permanent.

G. NAME - UPPER/LOWER

The PRONTO naming scheme relies on a look up table implemented as an association list under LISP. Design changes were made to this module due to the present system's inability to convert uppercase to lowercase, and vice versa. Permanent upper case involves finding the target text, copying this text and replacing current text through list surgery. Lowercase involves the same process. The text named by a change from lowercase to uppercase, is determined by the preceding textual class delimeter. The command interpreter determines the name's scope. No additional changes need be made once the new text replaces the old text.

H. PCM - Move Program Cursor

Program cursor moves control command execution. The program cursor may move either backwards or forwards within its scope. Program cursor moves require temporary forward and backward pointers to be established. Like find, PCM wraps around within its scope and does not move unless a move can be made.

Program cursor moves can be made at both the execute level and the

subprogram level. At the execute level, PCM's allow the user to execute subprograms. This is done in combination with an execute command.

CHAPTER V

PRONTO STRUCTURE

A. PREPROCESSOR

1. Design Requirements:

The main characteristics of PRONTO require abilities to both interpret text and to procedurally manipulate text. PRONTO, like LISP, is a symbol manipulation language that incorporates uniformity of program and data. Most implementations of LISP start as interpreters; all word processors are interpreters; PRONTO being a text oriented language naturally starts off this same way.

Word processors rely on the user to provide control, although some global features allow for internal context testing. For PRONTO to be a proper virtual machine, it must provide procedural control. The text and commands, in PRONTO, are part of the same text structure. The language primitives and functions are designed to be simply part of the text.

A complete PRONTO system is then:

Word Processor
\+ Command Sequences
\+ Command Interpreter

Command sequences are simply part of the text. The PRONTO interpreter is simply a text interpreter, providing command interpretation using the

text's natural structure as its guideline.

2. Functions

The goals of PRONTO preprocessor are to provide:

(1) a simple and convenient means of inputting characters, and

(2) an easily accessible storage structure, capable of retrieving these characters as text.

Several features must be included for handling named text descriptions, text hierarchical classes, syntax and procedurality. Although implemented using LISP, the goal is to define in terms of PRONTO all processing requirements and structures.

The preprocessor is an interpreter designed to accept text a character at a time. Each character is then analyzed and stored in an easily accessible structure, a simple text buffer. There are only two classes of characters to analyze:

(1) natural text separators, and

(2) the remaining alphanumeric characters.

For convenience and logic, normally non-printable separators such as end-of-paragraph, end-of-chapter, end-of-text, etc. are required to be explicitly input. These separators appear in natural text through skipped lines or skipped pages or like means. Storage requirements are kept simple by representing these separators through single characters.

The PRONTO preprocessor (fig. 3) serves as a monitor to control character input. The flow control follows two generally simple paths:

(1) processing of natural separators, and

(2) processing of all other non-separator characters.

For handling the remaining text features, simple conventions are

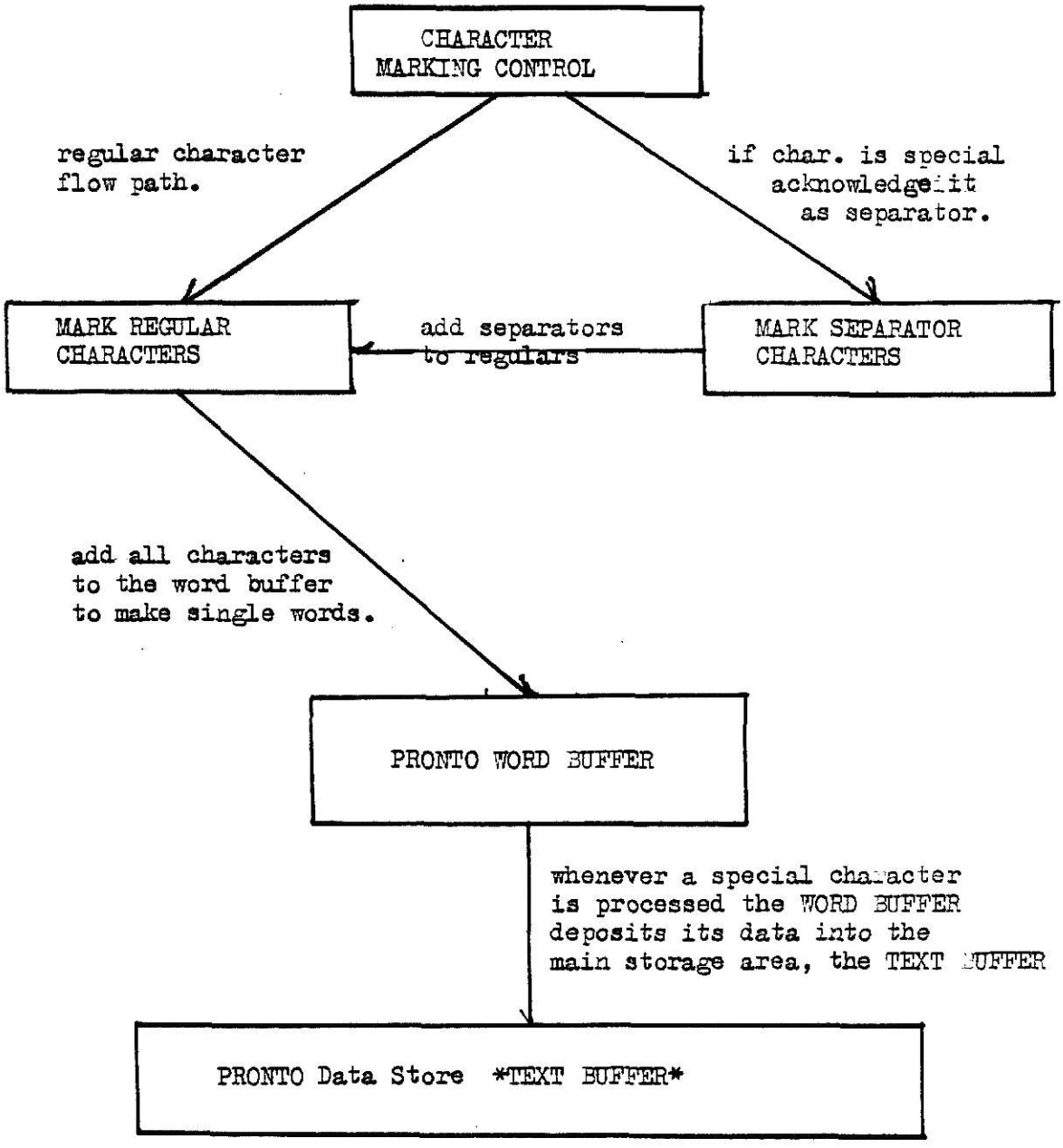


fig 3. PRE-PROCESSOR FLOW.

added to PRONTO'S syntax. The conventions used allow for flow along the described processing paths.

(1) Named Text Descriptions - These USED character groups are handled by adding a new unique separator to the natural text separators. PRONTO incorporates this separator as an explicitly input character, much like the other non-printing separators. Logically, this is a syntactic convention, but from a natural text standpoint, this USE distinction is itself a separator. Natural text incorporates highlighting, italics, bold face, underlining or a similar method to distinguish this text. Flow control follows the path for all separators.

(2) Procedurally - Keywords are used in most conventional programming languages. PRONTO introduces separators for:

(a) Comments -- This implementation incorporates the usage of angle brackets <comment> , treating the angle brackets as a special case of the separators.

(b) Parameters -- Single quotes denote the use of parameters in PRONTO. Flow control on input follows the separator's path.

(c) Primitives and Name References -- Double quotes are used as separators for reference to USED text, or to reference text classes.

EXAMPLE 1: "eos"

"sentence"

Double quotes also delimit PRONTO function primitive parameters.

EXAMPLE 2: Find "hat", where hat denotes the object for the primitive to perform its action.

(3) Text Classes -- Procedurally, a simple lookup table is used to handle the order of precedence of the class separators.

EXAMPLE 1:

CLASS	FOLLOWED BY
Word	Space
Phrase, Clause	Comma, colon or semi-colon
Sentence	Period
Paragraph	Eop (*Special Character*)
End of Text	Eot (*Special Character*)

The hierarchical strength is depicted as the table flows downward.

EXAMPLE 2: Find "eoph"

(1) This is a very short sentence.

(2) The second sentence, unlike the first sentence, contains a clause.

If the cursor is somewhere within the first sentence, the find operation will stop at the higher text class. In this case, the end of sentence will halt the find.

Name descriptions are not contained as part of the hierarchy, although it should be noted that named/used text contains not only the name separator, but natural text separators surround the name as well.

EXAMPLE: This SPECIAL sentence contains a named word.

*Where SPECIAL is all capitals is delimited by blanks on both sides.

Storage conventions adopted by the PRONTO preprocessor are kept simple, allowing for easy regeneration of input and minimal text character storage. Text is stored by:

(1) keeping only the highest order separator needed to regenerate other contiguously input separators, and

(2) single characters are formed into single words before being placed in PRONTO's permanent text store.

EXAMPLE:

1. Desire Output

CHAPTER ONE

THIS IS A SECTION

This is the first sentence in the first paragraph. This is a short paragraph.

The second paragraph is still shorter than the first one.

2. Input Would Be

(eochap) (eoname) chapter one (eoname) (eosect) (eoname) this is a section (eoname) (eopara) this is the first sentence in the first paragraph. This is a short paragraph. (eopara) the second paragraph is still shorter than the first one. (eot)

The input characters in brackets represent explicit non-printing characters. Each word separated by a blank, represents the word and its implicit end of word terminator.

3. Storage

The text and separators are stored as a simple linear list. Names are stored as sublists within the larger list. Parameters, single quotes as delimiters, within a named piece of text are stored as further sublists of the name sublist.

OUR EXAMPLE:

(eochap (chapter one) eosect (this is a section) eopara this is the first sentence in the first paragraph. This is a short paragraph eopara the second paragraph is still shorter than the first one eot)

All text in PRONTO, is surrounded by a header and tailer, which depict the text class to which pieces of text belong. Natural text provides this structure be it implicitly, through line skips or indentation or similar methods, or explicitly with periods, blanks and commas. PRONTO, like all word processors, makes all text and text classes explicit.

The PRONTO preprocessor uses two buffers to manipulate and store data. The first buffer is used to build words between separators; the second buffer (fig. 3) is to store the separators and formed words. Procedural matters are not considered until all input has been stored in the permanent text store.

B. THE COMMAND INTREPETER

The command interpreter is designed to process PRONTO function primitives from both execute level and subprogram level. The command level allows the user editor access to PRONTO's data store. The subprogram level is contained within the PRONTO store. Subprogram execution must

initiated from the command level.

The command level functions as a PRONTO editors. Available commands are:

1. Copy -- Copies words, sentences, paragraphs or any other

member of the hierarchical text classes to the current program cursor. Copy is used, normally, in conjunction with a move cursor command.

2. Move -- Like copy except the block of data moved is replaced by a null character. The MPC command supplies the destination for the move. Data may be deleted by moving to an unused storage area.

3. Add -- The add command is used to transfer control back to PRONTO's preprocessor. Normal preprocessor text input takes place. To Add to text without leaving the editor, the copy command is invoked. The copy command processes original data as well as existing data.

4. Find -- At the editor level searches for desired text or text class. Upon successfully finding the text the active cursor is moved to that location. Find is still a success or failure function at this level. However, the move cursor command is automatically invoked and cursor movement follows upon success. Currently find only works for subprogram levels.

5. Display -- Uses the subprogram find to locate the starting location for the display command to use. Upon successfully finding the data, the desired text is displayed in easy to read, pretty print format. At present, only design work has been completed to print the pretty-print version on the printer. The same pretty-print can, however, be printed through the underlying LISP machine.

6. Open -- Reads in an entire file using the preprocessor. After completion of the read, the level remains the same, the editor commands are all still accessible.
7. Close -- Prompts the user for a name to fill with, and then proceeds to write to that named text all characters explicitly within the class of the close parameter. 'Close text' will terminate the entire PRONTO session.
8. PCM -- The program cursor move command is invoked to move the cursor for: displaying, copying to, or moving to desired location of action, for which all editor commands do thier work.
9. Execute --The execute command attempts to process the current cursor location. Subprogram control is attained when the cursor is placed on an appropriate PRONTO program.

All editor-command level functions are applicable at the subprogram level, with the exceptions of add and execute. Add and execute cause control to be shifted only from within the editor. The subprogram interpreter behaves much differently from the editor. The subprogram level in PRONTO consists of a full PRONTO program making strictly subprogram calls. Each subprogram may in turn invoke other subprograms. Strict scoping rules must apply. The design of the subprogram level is:

1. Program Calls -- From the PRONTO program, the first subprogram call is initiated. Subprograms are named pieces of text within the PRONTO store. However, subprogram calls are not necessarily named calls. The

PRONTO program is simply a series of subprogram references. The references are by name, not value, and usually refer to a named subroutine. The command interpreter must perform a simple match between unnamed subroutines. Since named text is special, a special buffer is maintained holding the locations of all named text. The primitive function find with option "subroutines", is issued to search linearly the special buffer area. Upon successful match, program control is shifted to the location designated by the resulting pattern match.

2. Program Execution--The resulting pattern match ignores all parameters in its search. Parameters are determined within the subprogram. These parameters are classified by type in the program calls, although, reference to parameters in the body of the subprogram is made by position.

EXAMPLE:

```
PRONTO PROGRAM
Sub 1 'sentence'
Sub 2 'word' to 'word'
*Position of parameters is the important aspect.
All references within subprograms are by position.
i.e. 'first sentence'
or   'second word'
```

Scoping rules are dynamic, but could be explicit if so indicated by the parameter reference.

EXAMPLE: Find 'word 1' (in sentence 1 (in paragraph 2)).

The scope is the current named piece of text where the program cursor is active. The entire scope could, in theory, be allowed to further change within this scope by allowing each text hierarchical class to denote a different scope. The initial design allows for scoping only named areas of text, thus allowing further potential for future growth. The scope may be changed by moving the program cursor to a named piece of text. Normal program cursor moves disallow this, because movement is usually made within the current scope. A set scope command can be employed, to allow shifting from one USED text area to another.

Automatic scope changes take place upon subprogram calls. Since each subprogram is a USED block, transfer of control to new blocks emits a new scope. The old scope is maintained by saving a second cursor in the scope of text referencing each new subprogram call. This allows for conventional programming to take place while maintaining the use of the natural text structure.

C. POST PROCESSOR

The PRONTO post processor is the preprocessor in reverse. The preprocessor does all the difficult work of syntax and name checking. The post processor simply writes the PRONTO text buffer to a book (a file) for system storage.

The post processor does not pretty-print the file, nor does it

keep all characters in the text store. Further storage optimization takes place by throwing away any excess characters the user has manually put in that the preprocessor could have generated itself. The post processor stores the data in minimal form required for the preprocessor to regenerate the user file.

CHAPTER VI

PRONTO MODULES

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

PRE_PROCESSOR

Calling Functions: PRONTO

Parameters: EXPLICITLY - NONE
IMPLICITLY - CHARACTER

Functions: MARK
DISPLAY

FUNCTION DESCRIPTION

GLOBAL BUFFER *COMMAND* IS INITIALIZED TO "INPUT". THE PARAMETER CHARACTER IS ISSUED TO ACCEPT THE FORTH COMING INPUT.

CASE *COMMAND* OF:
INPUT, USED or MENTIONED : MARK
EXECUTING : DISPLAY

This function monitors text input, Processing is transferred between the PRE_PROCESSOR and the EXECUTION level. The processor monitor does not analyze characters, but issues space for new input and monitors whether or not the USED/MENTIONED distintion is on or off. The MARK function returns all character information to *command* for PRE_PROCESSOR analysis.

MARK(CH)

Calling Function; PRE_PROCESSOR

Parameters: CHARACTER

Functions Called: GET_RDMACRO
MARK_CHARACTER
MARK_SPECIAL

FUNCTION DESCRIPTION

The CHARACTER parameter is in fact a new character. The CH receives the value of the character "READ IN" from the Text Input Stream(TIS).

```
CASE CH of:
     "SEPARATORS"; MARK_SPECIAL
     "COMMENT":    GET_RDMACRO
     Otherwise:
                   MARK_CHARACTER
```

This function performs the actual system level "READ". The new CHaracter is then analyzed to determine which of the text flow paths to follow. Processing occurs through the subroutines called, USED/MENTIONED distinction is monitored, and results are passed to the PRE_PROCESSOR.

MARK_SPECIAL(T_TYPE)

Calling Functions: MARK

Parameters: T_TYPE ;Text type

Functions Called: GENERAL_APPEND
MARK_CHARACTER

FUNCTION DESCRIPTION

```
If *COMMAND* is USED & T_TYPE Not 'eow'
    MARK_CHARACTER(T_TYPE)
ELSE
    GENERAL_APPEND    ; clear out word
                      ; buffer
    Add the Word Buffer to *TEXT*
endif
Return to MARK & set up for next Input.
```

This function processes all PRONTO *SEPARATORS*. Named text, sets the *COMMAND* to USED, the *COMMAND* remains USED until the next "eoname" character is received, Buffer control takes place in the form of emptying the word buffer through the call to GENERAL_APPEND, the *TEXT* buffer is updated to include the new word and the succeeding *SEPARATOR*.

MARK_CHARACTER(NEWCH)

Calling Functions: MARK
MARK_SPECIAL

Parameters: NEWCH

Functions Called: NONE unless *COMMAND* is USED
If USED Then UPS
-UPS simply replaces the NEWCH with its uppercase equivalent.

FUNCTION DESCRIPTION

If *COMMAND* USED & Not Empty Word Buffer
UP(NEWCH)
endif.
Add all NEWCH's to Word Buffer, ignoring (eow) blanks. Blank characters are used to empty the word buffer, except when USED in Named Text. USED text does not dump the Word Buffer until an "eoname" *SEPARATOR* is encounterd.

Each NEWCH is simply put into the Word Buffer until MARK_CHARACTER is called by MARK_SPECIAL. When *SEPARATORS* are encountered, *TEXT* is updated. Text Cursor movement is monitored at this level, mostly for future full screen implemetation. The *TC* (text cursor) points towards each new input, with the exception of blanks, which are not stored.

GENERAL_APPEND

Calling FUnctions: MARK_SPECIAL
COMMENTS
GET_RDMACRO

Parameters: NONE

Functions Called: Used-or-Mentioned

FUNCTION DESCRIPTION

Invoke LISP "IMPLODE" to make an Atom(word).
Add the new word;
If new word is USED Then
Used-or-Mentioned
Else
Add word to *TEXT* and Empty
the Word Buffer(*BUFFER*).

USED-OR-MENTIONED?

Calling Functions: GENERAL_APPEND

Parameters: NONE

Functions Called: NAME-IN-BUFFER

FUNCTION DESCRIPTION

If *COMMAND* is MENTIONED Then
make *BUFFER* an individual List for
Internal Storage of PRONTO.
Else
NAME-IN-BUFFER

NAME-IN-BUFFER

Calling Function: USED-OR-MENTIONED

FUNCTION DESCRIPTION

This function adds another LIST level to
the contents of the *BUFFER*. PRONTO stores
USED text as SUBLISTS within *TEXT*.

COMMENTS

Calling Function: GET_RDMACRO

Parameters: NONE

Functions Called: GENERAL_APPEND

FUNCTION DESCRIPTION

```
GENERAL_APPEND    (* dump *BUFFER* to *TEXT* *)
Initialize temporary buff. with comment
delimeter " ".
WHILE  character NOT comment delimeter DO
       READ in character
       ADD character to local temporary buffer
           when blank character is reached ADD
           local buffer to *BUFFER*.  CLEAR
           local buffer.
 EOW.   (* end of while *)
 ADD *BUFFER* with comment, to *TEXT* and
 clean out *BUFFER* for new input.
```

This function handles all buffer manipulation and all input once the initial comment delimeter is encountered. Reads are separated to prevent PRONTO from mis-interpreting the special text, provided through comments.

GET_RDMACRO(CHAR)

Calling Function: MARK

Parameters: CHAR

Functions Called: GENERAL_APPEND
COMMENTS
PARAMS/PRIMS
EOT

FUNCTION DESCRIPTION

```
CASE char OF:
      comment delimiter:  COMMENTS
      Parameter delimiter: PARAMS/PRIMS
      Primitive delimeter: PARAMS/PRIMS
      End of Text Marker:  EOT
end case.
If CHAR is "eot" Then
   Return "eot"
Else
   GENERAL_APPEND
```

EOT(END_TEXT)

Calling Function: MARK

Parameter: END_CHAR

Functions Called: GENERAL_APPEND

FUNCTION DESCRIPTION

This fuction closes out the *BUFFER*, GENERAL_APPEND
The *text* buffer is complete, all *COMMANDS*
are updated.
TC is set to the start of the *TEXT* buffer.
PC is set to the start of *TEXT*.
COMMAND is set to "Executing", *SCOPE* is
globally established, and set to *TEXT*.

PARAMS/PRIMS

Calling Function: GET_RDMACRO

Parameters: NONE

Functions Called: NONE

FUNCTION DESCRIPTION

```
While NOT Parameter or Primitive
          delimiter DO
      Read character
      If character Used, UPS

         Add character to local
             temporary buffer,
             excess blanks are
             ignored.
         If local buffer full Then
             Add local buffer to *TEXT*,
                 reinitialize local
                 buffer and proceed.
end WHILE.
```

This function simply treats parameters and primitives like they are comments, with one major exception. All USED text and Params/Prims inside of other Used text are made into sublists. Params/Prims within other Named text are nested two lists deep.

Ex: (other text "eon" (THIS (word) IS NAMED)

- where "eon" is stored as the first level sublist brackets, and the inner list represents a parameter.

DISPLAY(TEXT_TYPE)

Calling Functions: PRE_PROCESSOR
GET_EXEC_COMMAND

Parameters: TEXT_TYPE

Functions Called: F_SCOPE
PARAMS_OR_PRIMS
DISPLAY_IT
GET_ITS_VALUE

FUNCTION DESCRIPTION

```
If TEXT_TYPE not a list, make it a list.
CASE TEXT_TYPE OF:
     If TEXT_TYPE length more than one word
        Then GET_ITS_VALUE
     First character equal
        parameter/primitive: PARAMS_OR_PRIMS
     Otherwise
              TEXT_TYPE
end case.
DISPLAY_IT
Return a blank.
```

This function is designed to print out its parameter, whether it is part of the *TEXT* or not. Control is passed upon determination of how to handle TEXT_TYPE. TEXT_TYPE as a member of *TEXT*. is displayed in a Natural Text format, with indentation, line feeds and punctuation supplied by PRONTO.

DISPLAY_IT(TEXT)

Calling Functions: DISPLAY

Parameters: TEXT

Function Called: F_SCOPE
DISP_NAME
DISP_TEXT_CLASS
DISP_TEXT_AS_READ

FUNCTION DESCRIPTION

```
CASE TEXT OF:
     *SCOPE*:  DISP_NAME(*SCOPE*)
     *NAMES*:  DISP_NAME(TEXT)
     *SEPARATORS*:  DISP_TEXT_CLASS(TEXT)
     Otherwise
             DISP_TEXT_AS_READ(TEXT)
 end case.
 DISPLAY_IT simply determines the type
 of TEXT to be displayed, and then gives
 control to the appropriate display handler.
```

This function analyzes to determine text classification for printouts to be displayed. DISPLAY_IT acts as an intermediary between the DISPLAY monitor function and text-type specific functions. The F_SCOPE routine is used here to reset the starting text cursor (*TC*) location, for the current *SCOPE*, thus allowing a view of current activity.

GET_ITS_VALUE(TEXT_LIST)

Calling Functions: DISPLAY
FIND

Parameters: TEX_LIST

Functions Called: PARAMS_OR_PRIMS

FUNCTION DESCRIPTION

```
If length of TEXT_LIST is one Then
    PARAMS_OR_PRIMS(TEXT_LIST)
Else
    we need to figure out what is
    between the front and rear delimiters.
    If there are no delimiters, then the
    routine returns the TEXT_LIST as it
    was received.  If there are delimiters
    they are removed by extracting the
    first and last characters, returning
    a new text-list.  Extensive checking
    is performed to insure that if delimiters
    are present, that both are there, front
    and rear.  As of yet, this function uses
    several nested LISP functions to work
    effectively.  For future PRONTO works,
    this function should be refined.
```

This function returns a useable text list to its calling function. The TEXT_LIST returned aids the finding and displaying of PRONTO Text.

DISP_TEXT_AS_READ(TYPE)

Calling Functions: DISPLAY_IT
FIND
CLOSE_FILE
DISP_TEXT_CLASS
DISPLAY

Parameters: TYPE (* text type *)

Functions Called: DISP_TEXT_AS_READ *RECURSIVELY*
PRINT_LIST_AS_READ

FUNCTION DESCRIPTION

Get length of TYPE. If length is one Then just print the TYPE out as is.
If length is greater than one, or if the TYPE is a list Then there are two paths to follow.

1. NON-LIST type, recursively call DISP_TEXT_AS_READ until length becomes zero
2. If the type contains a list, then get the inner list and call DISP_TEXT_AS_READ giving it the inner list. The recursive call backs out searching for any other sublists within the same structure.

This primitive subprogram may be used in place of the LISP print command, for displaying errors as well as displaying debug statements. For closing a file, this function simply types out the LISP-PRONTO storage area as is. Any number of parameters may be given to this process.

DISP_TEXT_CLASS(CLASS)

Calling Functions: DISPLAY_IT
PCM

Parameters: CLASS

Functions Called: PRINT_IT
DISP_TEXT_AS_READ

FUNCTION DESCRIPTION

```
If "eot" Then set *TC* to just prior
to end of text.
WHILE CLASS Not equal to next occurrance
      of equivalent or higher text class DO

      PRINT_IT(CLASS)
end while.
Return a blank
```

This function monitors the actual system print routine. Processing requires knowledge of the current *TC* and also requires knowledge of a text *SEPARATOR*. An iterative loop is employed to print text class pieces one at a time. Formatting of the actual print takes place in each individual call to the routine PRINT_IT. A simple look up table is maintained for determining text class hierarchy. Another simple look up table is maintained for text class *SEPARATORS* and there synonyms.

ex: "eos", "sentence" or "sent" are equivalent to a single "." (period,exclamation,question mark). Each class hasits own table.

PRINT_IT(TYPE)

Calling Function: DISP_TEXT_CLASS

Parameters: TYPE (* PRONTO *SEPARATOR* FOR PRINT *)

Functions Called: NEW_PAGE
TITLE
CENTER_60
DISP_TEXT_AS_READ
PRINT_ATOM
PRONT_SEPARATOR_PRINT

FUNCTION DESCRIPTION

CASE TYPE OF:

If List : DISP_TEXT_AS_READ
"EOT" : DISP_TEXT_AS_READ(End-of-Text)
Non-Separator: PRINT_ATOM
Separator : PRONT_SEPARATOR_PRINT

end case.

This function sets up for the initial formatting of PRONTO's *TEXT*. Each individual text type is separated out to allow individual treatment when printing. PRINT_IT performs basic monitor functions, by determining which class each character type belongs to.

NEW_PAGE

Calling Function: PRINT_PART
PRINT_END_BOOK

Parameters: NONE

Function Called: NONE

FUNCTION DESCRIPTION

Counts lines (*LINES* Buffer) and Skips to the top of a new page or screen.

CENTER_60(TAB-NAME)

Calling Function: TITLE

Functions Called: DISP_TEXT_AS_READ

FUNCTION DESCRIPTION

Counts to middle of page, divides TAB-NAME by two and counts that far backwards, placing the *TC* for next printing character.

TITLE(TEXT_TITLE)

Calling Functions: PRINT_EOSCT
PRINT_PART
PRINT_END_BOOK

Functions Called: DISP_TEXT_AS_READ

FUNCTION DESCRIPTION

Prints out the TITLE using the DISP_TEXT_AS_READ routine. Named text and parameters are printed out without ther internal brackets.

PRONT_SEPARATOR_PRINT

Calling Function: CLOSE_FILE
PRINT_IT

Parameters: NONE

Functions Called: PRINT_EOW
PRINT_EOS
PRINT_EOCL
PRINT_EOP
PRINT_EOSCT
PRINT_PART
PRINT_END_BOOK

FUNCTION DESCRIPTION

The *TC* or a temperary cursor, points to the next text class character to be printed.
CASE *TC* OF:

"EOS" type : PRINT_EOS
"EOW" type : PRINT_EOW
"EOCL" type : PRINT_EOCL
"EOP" type : PRINT_EOP
"EOSCT" type : PRINT_EOSCT
"EOPRT" type : PRINT_PART
"EOB" type : PRINT_END_BOOK

end case.

This function performs basic monitoring of text separator print modules. Each *SEPARATOR* is handled separately, providing for individual formatting. No parameters are needed, the print cursor supplies all needed location information.

PRINT_EOW

Calling Function: PRONT_SEPARATOR_PRINT

Functions Called: NONE

FUNCTION DESCRIPTION

Prints a blank between words

PRINT_EOS

Calling Function: PRONT_SEPARATOR_PRINT

Functions Called: STAB

FUNCTION DESCRIPTION

Prints a period, as separator by default, however it also prints other "eos" separators. After each period STAB - tabs two blank characters.

PRINT_EOCL

Calling Function: PRONT_SEPARATOR_PRINT

Functions Called: STAB

FUNCTION DESCRIPTION

Prints a comma by default, phrases are not handled separately. Each phrase or clause separator is followed by a single tab.

PRINT_EOP

Calling Function: PRONT_SEPARATOR_PRINT

Functions Called: STAB

FUNCTION DESCRIPTION

An automatic new line is inserted after a period. The first character of the next sentence is automatically capitalized.

PRINT_EOSCT

Calling Function: PRONT_SEPARATOR_PRINT

Functions Called: TITLE
STAB

FUNCTION DESCRIPTION

Prints a period followed automatically by two new lines. TITLE is called when the next *TEXT* item is USED.

PRINT_EOPART

Calling Function: PRONT_SEPARATOR_PRINT

Functions Called: NEW_PAGE
TITLE

FUNCTION DESCRIPTION

Prints a new page and closes the preceeding part with a period. TITLE is invoked as in PRINT_EOSCT.

PRINT_END_BOOK

Calling Function: PRONT_SEPARATOR_PRINT

Functions Called: NEW_PAGE
TITLE

FUNCTION DESCRIPTION

This function closes preceeding text classes. At present "EOB" is equivalent to "EOT" (end of text). The "EOB" or "EOT" on input causes processing to be shifted to the command level. DISPLAY will then, print the entire *TEXT*.

* IT SHOULD BE NOTED THAT ANY NUMBER OF TEXT LEVELS COULD BE USED, THESE ARE JUST A FEW USED IN THIS IMPLEMETATION.

FIND(TYPE)

Calling Functions: DISPLAY
DISPLAY_IT
DISP_TEXT_CLASS
DISP_NAME
COPY
MOVE
F_SCOPE
PCM
POST_PROCESSOR

Parameters: TYPE (* any text type *)

Functions Called: FIND_NAMES
FIND_IT
F_SCOPE
GET_ITS_VALUE

FUNCTION DESCRIPTION

```
If *COMMAND* TYPE Is F_SCOPE Then
    find locates current *SCOPE*.
CASE TYPE OF:

     *NAMES* (USED) : FIND_NAMES
     *SEPARATORS*   : FIND_IT
     Otherwise
            FIND_IT
 end case.
Return is "success or Failure"
```

This function is the heart of PRONTO processing. Control is maintained through a series of type checks resulting in specific FIND routines being invoked. This command, edit and subprogramming function is used in defining other PRONTO primitives. COPY, PCM, MOVE OPEN, CLOSE and DISPLAY all issue finds to locate *SCOPE*, *NAMES* and *SEPARATORS* (text classes).

FIND_NAMES(N_TYPE)

Calling Function: FIND
DISP_NAMES

Parameters: N_TYPE (* gives the *NAMES* stored *)

Functions Called: NONE

FUNCTION DESCRIPTION

```
If *TEXT*, *TC*, *PC*, or *SCOPE* is
    the N_TYPE Then set *SCOPE* if
    N_TYPE is Name within these buffers.
*SCOPE* is determined in FIND, therefore the
search for names takes place only within that
*SCOPE*.
WHILE not found DO or "EOT" or end of *SCOPE*
      If next *PC* is a name Then
           Is name our name?   -- success
                               or -- failure.
      Else
           *PC* moves to next instruction
           Result of success, halts search
           *PC* is restored and further instructions
           continue.
```

This function searches the named (USED) text only. Processing other *TEXT* requires different FIND calls. Find-Names does not destroy any PRONTO buffers, but instead copies new buffers for local usage. Optimization of FIND_NAMES, would involve a more sophisticated *NAMES* buffer, using this buffer as a dynamic location name store. This implemetation keeps track of *NAMES* by existance.

FIND_IT(TYPE)

Calling Function: FIND

Parameters: TYPE (* non-named *TEXT* *)

Functions Called: F_SCOPE
HELP_FIND_IT

FUNCTION DESCRIPTION

Find current *PC*'s *SCOPE*, F_SCOPE
Save all PRONTO buffer areas.
Invoke HELP_FIND_IT, to actual search
for TYPE.

HELP_FIND_IT(WHERE?)

Calling Function: Find_IT

Functions Called: GET_ORDER

FUNCTION DESCRIPTION

```
While NOT Found DO
     CASE WHERE? OF:
          Current temp *PC* : found is now true
          "EOT" or "EOB"    : Print error message
                              'FAILURE'
                              Return nil
           Otherwise
                    Get next *PC* and Search
```

F_SCOPE

Calling Functions: FIND
FIND_NAMES
DISPLAY
DISP_NAMES
PCM
FIND_IT

PARAMETERS: NONE

Functions Called: NONE

FUNCTION DESCRIPTION

```
Set up temporary cursor at start of *TEXT*
WHILE  cursor NOT  *SCOPE* DO
     " Search Text "
     If cursor EQUAL *SCOPE* Then
         " Mark it Found "
     Else
         "Continue linear search "
          cursor -- cursor PLUS one
 end while.
 *  If not found then *SCOPE* does not change.
```

This function performs a linear search of the entire PRONTO *TEXT* buffer. The search is optimized by looking only for named *TEXT*, as this version of PRONTO allows scoping only for USED *TEXT*. The future goal, however is to provide scoping by text class. The development stages considered this aspect, but it was decided to forgo this in favor of an over all view of more features.

FIND_SUBS( )

Calling Functions: FIND
GET_EXEC_COMMAND

Parameters: Explicitly - none
Implicitly - *BUFFER*

Functions Called: FIX_S_NAME
EXEC
TRY_TO_MATCH

FUNCTION DESCRIPTION

"Search" the *NAMES* Buffer for single word match. If a single word match is successful try to match the remainder of the name.

EX: *NAMES* (THIS IS TEXT)

"SEARCHING FOR" THIS WAS TEXT

single word match is successful with "THIS" TRY_TO_MATCH -- FAILS, becuase of the mismatch on the second character.

EX2: *NAMES* (THIS (IS) TEXT)

"SEARCHING FOR" THIS''was'_TEXT

single word match succeeds, TRY_TO_MATCH also succeeds. Parameter matching is by substitution, thus the only time the parameter is of significance is when in the subprogram itself.

Subprograms are executed, upon successful match through PRONTO's Exec handler. THE EXEC handler simply determines processing of edit level commands versus Subprogram (PRONTO PROGRAMS) level commands.

PCM(WHERE?)

Calling Functions: GET_EXEC_COMMAND

Parameters: WHERE? (* Destination and Direction *)

Functions Called: DISPLAY
FIND

FUNCTION DESCRIPTION

The Parameter WHERE? is first analyzed
to determine, destination (BY TEXT CLASS)
and direction (Forward or Backward). Both
options do not have to be present, defaults
are "WORD" & "FORWARD".
If WHERE? is (member of Text Class) and direction
is Forward or default, Then
FIND "Text Class"
If found Then set *PC* to SEARCH Location.
Else DISPLAY "NOT FOUND"

Else
Mark search start,
F_SCOPE, (* puts cursor behind current local *)
Scan Forward until matching Text Class, Mark it
Scan Forward until another occurrance of match
unless we pass the starting MARK.
The last Mark before the start now holds the
reverse pointer.

endif.

This function employs previously defined PRONTO primitives
to move the *PC* to a desired location.

CHAPTER VII

AN EXAMPLE PRONTO PROGRAM

The PRONTO VM defines a rather primitive language. Its main use is intended to be the implementation of high-level description languages. However, a simple example will illustrate some of its potential. The chosen algorithm is a simple spelling checker. The dictionary resides in a book (file) called DICT and the text to be checked in a book called TEXT. The program is written out as it would appear on the screen rather than giving its rather confusing internal representation.

SPELLER

```
    Prepare the dictionary "DICT".
Read in the text (to be checked) "TEXT."
For each word in "TEXT" check against "DICT".
End.
PREPARE THE DICTIONARY 'word'
    Open book 'first word', find "eochap" <the end of this program>, copy
'first word'. <the dictionary is now appended to the program>
READ IN THE TEXT 'word'
    Open book 'first word', find "eochap"<the end of this program>, copy
'first word'.  <the text is appended>
FOR EACH WORD IN 'word' CHECK AGAINST 'word'
```

```
<DICT is organized as a paragraph of sentences, each one of which
has the entry as its first word >
Find 'first word', find "eoname".  Find "eopara" in 'character', end on
success.

    Copy next, look for next in 'second word', pcm 'sentence' on
success, print next not found.  Find next (in 'first word'), delete 'word',
find "eoword", find "eochar", pcm 'paragraph' back.  NEXT **n**.
LOOK FOR 'word' IN 'word'

    Find 'second word', find "eoname".  Find 'first word' in 'sentence',
end on success, find "eosent", find "eopara" in 'character', pcm 'sentence'
back on failure, end failure.
PRINT 'word' NOT FOUND

    Display "word not found", display 'first word', display "eoline".
```

REFERENCES

1. Charniak, Eugene; Riesbeck, Christopher K.; McDermott, Drew V., ARTIFICIAL INTELLIGENCE PROGRAMMING , Lawrence Eilbaum Associates, Publishers, Hillsdale, New Jersey, 1980.

2. Wembley, D. W. and Nagy, G.'Behavioral Aspects of Text Editors' Computing Surveys, Vol. 13, No. 1, March 1981.

3. Allen, John, ANATOMY of LISP, McGraw Hill, New York, 1978

4. Barr, Avron, and Feigenbaum, Edward A., The HANDBOOK of ARTIFICIAL INTELLIGENCE, Volume 1, Heuris Tech Press, Stanford California.

5. Hartley, R. T. PRONTO Programs and Examples.

THE DESIGN AND IMPLEMENTATION OF PRONTO PROCESSOR FOR NATURAL TEXT ORGANIZATION

by

Steven Michael Anderson

An Abstract

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1983

ABSTRACT

The following implemented virtual machine design, is intended to further develop research in the area of descriptive languages. The only data structure supported is natural organized text. The design incorporates word processing and basic editor functions to procedurally manipulate text.

The machine is designed and implemented using LISP, the goal however, is to define PRONTO (Processor for Natural Text Organization) in terms of its text handling. PRONTO naturally incorporates many features found in LISP, uniformit of Program and data between internal and external representations and hierachic text capabilities, to name just two. PRONTO is designed to remove some of the tedious aspects of LISP, such as syntax and structure. LISP provides no easily accessible data structures, without extreme manipulation of its primitives. PRONTO is aimed at supplying uniformity of data and simplicity of structure, through its own set of basic primitives.

PRONTO is a low level machine type language. The PRONTO machine incorporates word processor and conventional programming primitives. Programs, in PRONTO, are written as text to be procedurally and texturally manipulated. The initial implementation, is designed to be enough to show capabilities of designing such a virtual machine.