A Study On Structure-Oriented Text Editing In The Interactive Computer Systems*

Koh Jae Jin · Wu Chi Su · Lee Jung Tae
Dept. of Computer Science

(Received September 1, 1983)

〈**Abstract**〉

Editing tasks are the most frequent daily tasks we do in daily life. It is the task that changes the state of some target entity, whether we do it by computer or other tools. This paper examines computer based interactive editing systems, first on general systems and next on structure-oriented text editing systems. This paper defines terms, and provides a comprehensive features of editing systems.

This paper provides user views of the editing process, and suggests system views of the editing process. Some historical perspective is presented and the functional capabilities of editor are explained and discussed. We emphasize on user-level rather than implementation level considerations.

This paper also presents the feature of the structure-oriented text editing systems which aims at a generalized approach to data editing in interactive computer systems. The structure-oriented text editor has the ability to handle hierarchical structures with screen-oriented text editing facilities.

대화형 콤퓨터에서의 구조지향적 문장편집에 관한 연구

고재진 · 우치수 · 이정태
전 자 계 산 학 과

(1983. 9. 1 접수)

〈요 약〉

편집작업은 우리들이 일상생활에서 가장 빈번히 하는 것이다. 이 작업은 컴퓨터로 하든, 다른 도구로 하든, 어떤 대상이 되는 자료의 상태를 변경하는 작업을 말한다.

이 논문은 컴퓨터를 기초로 한 대화형 편집 시스템에 대해서 연구하고, 처음에는 일반 시스템에 대해서 서술하고, 다음 부분에서는 구조지향적 문장편집 시스템에 대해서 서술한다.

이 논문에서는 용어를 정의하고, 편집시스템의 중요한 부분에 대해서 자세하게 설명했다.

이 논문은 편집과정을 사용자가 보는 입장과 시스템이 보는 입장에서 서술했다. 편집기가 발달해온 역사적 과정을 요약해서 서술했고, 편집기의 기능에 대해서 서술했다.

우리는 편집기를 제작하는 관점에서 보다 사용자의 입장을 중요시하는 관점에 역점을 두었다.

이 논문은 대화형 컴퓨터 시스템에서 데이타 편집을 통일적으로 할 수 있는 포괄적인 구조지향적 문장 편집 시스템의 중요한 양상에 대해서 서술했다. 구조 지향적 문장 편집기는 화면에서 문장편집이 가능한 장

* 이 논문은 1982년도 문교부 학술연구 조성비에 의하여 연구되었음.

치를 이용해서 계층적 구조를 가진 문장을 쉽게 편집할 수 있는 능력을 갖고 있다.

I. Introduction

Editor is a program that edits data and programs in computer storage.

The interactive editor is an editor that operates interactively for editing data and programs. Nowadays the interactive editor becomes an essential part of computer systems. The functions of the editor are the creation, addition, deletion, and modification of data and programs.

The interactive editor will become key components of the office automation systems, because of it's characteristics to edit files, manuscripts, messages, lists, reports, etc. Knowledge workers, whether they are authors, composers, researchers, teachers or doctors, think the editor an important tool they use daily life. There has been little researches to formalize terminology, or to make a framework for analyzing editing systems. This paper is a comprehensive introduction to text editing, especially on structure-oriented text editing.

II. The internal structure of the editor

Editors have an architecture similar to Fig. 1. The interaction language processor accepts input from the input devices, lexically analyzes and tokenizes the input stream, syntactically analyzes the stream of tokens, and invokes the appropriate semantic routines At the syntactic level, the interaction language processor may generate an intermediate representation of the desired editing operations. The semantic routines invoke traveling, editing, viewing, and display.

With the viewing and displaying, there need not be a one-to-one relationship between what is "in view", that is, what is displayed on the screen currently, and what can be edited. In editing a document, the current editing pointer determines the start of area to be edited.

The current editing pointer can be set or reset by the user with a traveling command.

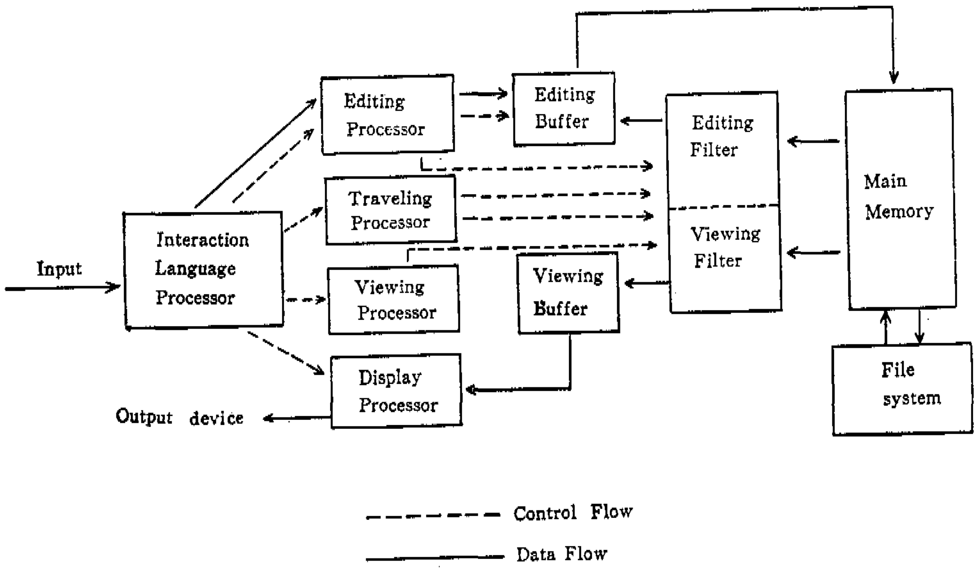

Fig. 1. The architecture of the editor

The traveling processor performs the setting of the current editing and viewing pointers. The editing filter filters the document to generate a new editing buffer based on the current editing pointer as well as the editing filter parameters.

The filter parameters provide such information as the range of text that can be affected by the operation, for example, the "current line" in a line editor or the "current screen" in a display editor.

Editing buffer is a filtered subset of the document data structure. The current viewing pointer determines the start of the area to be viewed.

When the display needs to be updated, the viewing filter filters the document to generate a new viewing filter parameters.

The viewing filter parameters provide such information as the number of characters needed to fill the display and how to select them from the document.

The viewing buffer is then passed to the display processor, which maps it to a window or viewport, a rectangular subset of the screen to produce a display.

The editing and viewing buffers can be related in many ways. They may be identical, disjoint, partially overlapping or properly contained in one another.

Mapping viewing buffers to windows that cover only part of the screen is useful for editors on modern, high-resolution rastergraphics-based work stations, as they allow the user to examine and to interact with multiple views, for inter-and intra-file editing and "cutting and pasting" The notion of multiple viewing buffers and multiple windows showing differing portions of the same or multiple files at the same time on the screen is designed into modern editors.

The viewing buffer-to-window mapping is accomplished by two processors of the system. The viewing processor formats an ideal view, expressed in a device-independent intermediate representation. The display processor takes this idealized view and maps it to a physical output device in the most efficient manner possible.

Much research is concerned with optimal screen updating algorithms that compare the current version of the screen with the following version and, using the basic capabilities of the terminal, write only those characters needed to generate a correct display.

Device-independent Input/output promotes interaction language portability. Many editors make use of a terminal control database. The processors communicate with a user document on two levels: in main memory and in the disk file system. In modern systems, the editor maps the entire file into virtual memory and is letting the operating system perform efficient demand paging.

In systems without virtual memory, editor paging routines are required. These read in one or more logical chunks of a document, called, pages into main memory, where they reside until a user operation requires another piece of document. Documents are often represented not as sequential strings of characters, but in an editor data structure that allows addition, deletion, and modification with a minimum of I/O and character movement.

The computing environments in which editors operate are timesharing, stand-alone, and distributed.

The timesharing editor must operate swiftly within the context of the load on the computer's processor, primary memory, and secondary memory. The editor on a stand-alone system must have access to the functions provided by a small local operating system.

The editor in a distributed resource-sharing local network must run independently on each user's machine and must contend for shared resources such as files.

The intelligent terminals have their own microprocessors and local buffer memories in which editing manipulations can be done.

Ⅲ. Structure-Oriented Text Editing

It is important in interactive computer systems that one can edit text data.

The editor is a program that are performing editing functions in computer systems.

There has been many unified approaches to the editing task. Handling hierarchically structured text was considered an important part of the editor.

This section is primarilly written for manipulation of hierarchically structured texts.

Text editing is the most frequent task to be done, so it has to be very efficiently done.

The users model must be simple. The first editors were very simple, often run through cards.

Since terminals were put on line, a rapid development of text editors took place. By the mid sixties, some text editors had full programming language capabilities[DAM 71].

The idea of this section came from taking the inherent structure of texts into account. This means that we can have several view options of a document.

For example, one can choose to display just the skeleton, perhaps the section headers, or look at everything in some paragraph. The most editor today are streamlined to run on video terminals. The current part of the text is kept on screen and every change on the screen immediately updates the mirrored text.

Nowadays, editors have been line-oriented or treated the text as string of characters.

Natural-language-like commands of the editor are easier to learn [LED80], but the commands must be very simple, keeping typing over head at a minimum [CAR80].

1. The K editor

The K editor is a structure-oriented text editor. The idea of this editor is to superimpose a tree structure into the text. The tree resembles a table of contents. However changes in the tree structure will cause the text parts to be correspondingly reordered. Selection of text part is made by traversing the tree structure using menus produced from the current structure. When a leaf is reached text editing mode is entered. The editor works through cooperation of a tree editor and a text editor.

The K editor has two types of nodes. Those are tree nodes and text nodes. A text node is a sequence of characters. A tree node is a list which has a head and a sequence of zero or more references to other nodes, its subnodes. The subnodes can be text nodes or tree nodes. If a subnode is a tree node it is the root of a subtree. In K editor the node head is a text node. We can put arbitrary information in the head. The node head can have informations such as node names, comments about the text.

In K editor we have a paragraph in each text node. The tree structure shows how these paragraphs from sections and chapters.

The node heads can have headers and formatting commands.

We can display the structure of a tree or the text in a node. If the current node is a tree node the first line of the head and the first line of each subnode is showed. The subnodes are numbered and these numbers can be used as arguments to a command.

The current node is the node that is showed on the screen. To traverse the different parts of the tree is to change current node.

Tree traversal is done by menu selection.

We can make the subnode a current node by selecting the subnode number. Some commands are needed for the user to walk around in a tree and get a general view of a structured document. There are commands to select the n th subnode, go up to root node, go one level up, go the next and the previous node on the same level in the tree. The idea of these commands comes from the structure editor in the INTERLISP programming system [TEI78] [SAN78].

There is a command to show several levels of the tree. The indentation of node numbers is used to indicate how the tree nodes are connected. If the current node is a text node the whole text is displayed. There is a command to show what the text would look like if all the nodes in the subtree were merged.

There are commands to delete a subtree, insert a node after current node, insert a node before current node, and to join current node with next node.

There are commands to change order between subnodes. There are commands to copy a subtree into another subtree. The creation of new tree nodes is done by grouping the nodes that shall be its subnodes.

There is a command to split a tree node into its subnodes. The edit command is used to change the information in a node. There must be the text editor for the text nodes. The head is edited if current node is a tree node. There are commands to communicate with the file system. The whole tree or a subtree can be saved on files with or without the tree structure. Files with structure can later be loaded as a subtree at any position in tree. Files without the structure information can be loaded as a text node.

The user can then split the text and build a structure over it.

The text editor have a part of the text presented in a window, and makes extensive use of control characters. The editor inserts normal characters at the place of cursor. For string replacement, it is made possible to change to command mode, where the command is visualized on a dedicated now. To create new text nodes, the text editor has two commands, one manual and one automatic.

The "new node"-command(control-N) splits the text at current position and inserts the first part of the text before current node while the second part is kept in the text editor. The "guess"-command on the command line is used to devide a text node depending on the characters in the text.

2. Internal Structure of K editor

The text nodes is referenced from a tree node or is on the free list. All texts are in one big buffer and a text node contains the length of the text and a pointer into the text buffer.

Since texts of variable length are inserted, deleted and changed, it is necessary to compact the buffer and reclaim unused space, All text nodes in the tree are linked together in the same order as they have space allocated in the buffer. Net text nodes are added in the end of the buffer. If there is not free space enough the buffer is compacted. Before editing a text it is copied to the end of the buffer. This makes it possible for the text to grow and it gives the user a chance to get back the old version of the node.

To delete a text node it is just to remove it from the list of active texts and put it on the free list. To compact the buffer, the list of active text nodes is went through once and the texts in the buffer are moved so they follow direct after each other.

A tree node is implemented as linked list. The first list element contains a pointer to the head followed by a list of zero or more subnodes. Each list cell contains two pointers and a type tag.

A tree structure is stored on a sequential text file with the structure with parentheses representing tree nodes and the structure of characters in text nodes written as decimal integers.

All text in the tree are written in depth first order. This external format for the structure is used to make the loading fast. The tree structure is built when the structure part of the file is read. Each new text node points to the place in the text buffer where the text later will be loaded. After the structure part is read the rest of file can be moved directly into the text buffer. Other file formats are program files where the structure information is strored in comments and text files without stucture.

Ⅳ. The manipulation of structured information

The K-system is for the manipulation of structured information. In interactive programming environments such as K-system, a programmer may design, implement, document, debug, test, analyze, validate, maintain and transport his programs. The K-system provides a programming team with tools for specifying a design, enforcing a programming methodology and verifying interfaces.

The kernel of K-system in a syntax directed eidtor in which every object is represented as an operator-operand tree, usually called an abstract syntax tree. For a given language, a tree grammar, called the abstract syntax of the languages, defines legal abstract syntax trees in that language.

These tree data structures are shared by several processors. The characteristics of this programming environment are interactive, programmable, language independent, and multilingual. Multilinguality means that several languages may be handled simultaneously and subtrees of the same tree may represent programs in different programming language.

1. The specification language

To add a new formalism, or, programming language, to K-system, it is necessary to write a specification language program that defines this formalism.

Since the Specification language is one of the formalisms that may be manipulated under K-system, it has been defined in itself.

The user who wants to add a new language to K-system will do so in the K-system programming environment. In a K-system session, the COMPILE command creates the tables for the formalism defined by the current specification language program.

Then, within the same session, K-system may be directed to use this new formalism as current language.

The first implementaation of the specification language under K-system has been obtained by bootstrap is handcompiling the specification language program that defines the specification language itself.

The specification language is one example of language whose compiler taken as input the K-system tree structure rather than the textual representation of programs.

A specification language program, say SP, that defines a formalism F, contains the following components.

(1) The concrete syntax of F

This is a set of rules to decide whether or not a given sentence belongs to F.

These rules will be used to create a parser for F.

(2) The abstract syntax of F

The syntax of trees to represent correct programs in the formalism F.

(3) The tree building functions

These functions indicate which tree corresponds to each component of the language.

The tree generator of F is created from these functions.

Each concrete syntax production is associated with a semantic action, called a tree building function.

When passing a program in F, whenever a reduction is performed by the parser, the associated semantic action is executed.

A tree is built and pushed onto a stack.

(4) The unparsing specifications of F

These specifications specify the converse connection, between, tree form and text form.

They indicate the piece of text associated with each elementary abstract tree.

The connection between text form and tree form is accomplished by two processors.

(1) the universal parser(constructor)

This builds a tree from the text form.

It is driven by tables compiled from the concrete syntax and the tree building functions.

(2) The unparser

This builds a text from an abstract tree.

It is driven by tables compiled from the unparsing specifications.

2. The syntax of a small programming language

This language is called ASPLE defined by its denotational semantics in [DON 78].

```
<program>::=begin<dcl-train><stm-train> end
<dcl-train>::=<declaration>
               |<dch-train><declaration>
<declaration>::=<mode><idlist>
<mode>::=bool
         |int
         |ref<mode>
<idlist>::=<id>
         |<idlist>, <id>
<stm-train>::=<statement>
               |<stm-train>; <statement>
<statement>::=<asgt-stm>
               |<cond-stm>
               |<loop-stm>
               |<transput-stm>
<asgt-stm>::=<id>:=<exp>
<cond-stm>::=if<exp> then<stm-train> fi
               |if<exp> then<stm-train>
                else<stm-train> fi
<loop-stm>::=while<exp> do <stm-train> end
<transput-stm>::=input<id>
                   |output<exp>
<exp>::=<factor>
         |<exp>+<factor>
<factor>::=<primary>
            |<factor>*<primary>
<primary>::=<id>
             |<constant>
             |(<exp>)
             |(<compare>)
<compare>::=<exp>=<exp>
             |<exp>#<exp>
<id>::=%ID
<constant>::=<bool>
             |<num>
<bool>::=true
         |false
<num>::=% NUMBER
```

3. The abstract syntax of a small programming language

The abstract syntax consists of operators and phyla. The operators label the nodes of the abstract trees. There are two kinds of operators: fixed arity and list operators. The arity of fixed arity operators is limited to 3.

Operators with arity zero are the leaves of abstract trees and represent atoms of the language.

Operators of fixed arity can have offsprings of different kinds, where as all offsprings of list operators must be of the same kind. The meaning of "kind of offspring" is formalized by the concept of phylum.

The phyla and non empty sets of operators. To each offspring position of an operator is associated a phylum. This phylum contains those operators that are allowed as had-operator of a subtree in this offspring position. Each occurrence in an abstract syntax tree is associated with a phylum that indicates what operators are allowed at this location.

This phylum depends on the father of this occurrence, and on the rank of this occurrence as a son of its father. To define an abstract syntax one must describe both operators and phyla.

For operators, this involves defining their arity and the phyla associated with their sons.

For phyla, this means defining which operators they include. To define the abstract syntax of a language, we must keep the following criteria in mind.

(1) Subtrees should correspond to semantically significant concepts of the language.

(2) The abstract structures should be close enough to the concrete syntax so that the user may understand and remember them easily.

The abstract syntax of a small programming language(ASPLE) is given below.

Operators are in lower case, phyla are in upper case. To define an operator, one gives the phyla of its sons, and to define a phylum one enumerates the operators that belong to it. The arity of an operator is equal to the number of phyla on the right hand side of the rule defining the operator.

Symbols +.. indicate that the operator is a list operator, for a list that cannot be empty, while symbols*... denote a list that can be empty.

A single phylum is given for all sons of a list operator.

The operator "program" is binary, its first son is an operator belonging to the phylum DECLS and its second son belongs to the phylum STMS. The operator stms is a non-empty list operator, the sons of which belong to the phylum STATEMENT.

The operators bool, int, and id are nullary operators and they will be leaves of abstract trees.

```
program──→DECLS STMS;
decls──→DECLARATION+...;
declaration──→MODE IDLIST;
idlist──→ID+...;
bool──→;
int──→;
ref──→MODE;
stms──→STATEMENT+...;
assign──→ID EXP;
ifthen──→EXP STMS;
ifthenelse──→EXP STMS STMS;
While──→EXP STMS;
input──→ID;
output──→EXP;
plus──→EXP EXP
times──→EXP EXP
equal──→EXP EXP
different──→EXP EXP
boolean──→;
number──→;
id──→;
comment──→;
comment-s──→COMMENT*...;
meta──→;
deref──→EXP;
and──→EXP EXP;
or──→EXP EXP;
DECLS::=decls;
STMS::=stms;
DECLARATION::=declaration;
MODE::=bool int ref;
IDLIST::=idlist;
STATEMENT::=assign ifthen ifthenelse
                while input output;
EXP::=plus times ID equal different
        CONSTANT AUX;
CONSTANT::=number boolean;
ID::=id;
COMMENT::=comment;
AUX::=deref and or;
```

4. translation from text to tree

An essential component of a specification language program is a list of rules that specify a concrete grammar with a translation of concrete text representation to abstract tree representation.

(1) Atomic trees

Atomic trees are built with the atom constructor. This constructor takes the name of an atomic operator as a left operand and the value of the atom as a right operand. For the following production

〈id〉; ; =%ID

when the parser makes a reduction using this production, it means that a token, accepted by the regular expression that defines %ID in the scanner, has been met in the text.

The generated tree must be an atomic tree with the operator id of the abstract syntax of a small programming language(ASPLE), and its value must be the string representation of that token. The function that builds such a tree is

id−atom(%ID)

In the specification language program we have the rule;

〈id〉; ; =%ID;

id−atom(%ID)

There are four rules in the specification language program that builds atomic trees:

〈id〉: : =%ID;

id−atom(%ID)

〈num〉: : =%NUMBER;

number−atom(%NUMBER)

〈bool〉: : =true;

boolean−atom('true')

〈bool〉: : =false;

boolean−atom('false')

(2) Fixed arity operators

For the following production,

〈mode〉: : =ref〈mode〉;

The tree generating function will be ref(〈mode〉) This means that the corresponding tree will consist of the unary operator ref with, as a son, the tree associated with the non-terminal 〈mode〉. For the following production,

〈factor〉: : =〈factor〉#＊〈primary〉;

The sharp sign, #, is the specification language forcing character. It has to precede every terminal which is a specification language keyword or punctuation-mark.

The tree generating function will be times (〈factor〉, 〈primary〉)

This means that a tree is built up with the binary operator times and, as first son, the tree associated with the nonterminal〈factor〉 and, as second son, the tree associated with the nonterminal〈primary〉.

Ⅴ. Conclusion

The editors offer a set of frequently used standard functions such as editing, traveling, viewing and display.

The K editor was designed prototype for a structure-oriented text editor. The structure handling mechanisms and the screen-oriented text editing primitives of the K editor were presented. The hierarchical structuring of text objects provides qualities for information handling of database management facilities. The flexible hierarchical structuring mechanisms is a very useful tool for organizing and maintaining larger text files or data collections. In the allocation of text to nodes a tree browsing interface based on menu selection sigficantly improves the access to node data. The K system is an abstract manipulation system at the core of a programming environment. The user corresponds to the system through the concrete syntax. The user can visualize his trees with unparsing, and conversely input his program text with parsing. The abstract syntax manipulation language should have a powerful procedure abstraction mechanism.

It is important that the K system can manipulate structured annotations, linked to the structure of the program, but conceived as separate entities. We envision a structure-oriented text editing system which unify the whole range of a programming activity such as design, development, documentation, debugging, maintenance and transport.

Ⅶ. References

〔BBN 73〕 Bolt Beranex And Newman Inc. TENEX text editor and corrector manual, combridge, Mass., Oct., 1973.

〔CAR 80〕 Card, s. et al, The Keystroke-Level Model for User Performance Time with Interactive systems, comm. ACM, vol. 23, pp. 396—410, July 1980.

〔COMS 67〕 Com-Share, Inc. QED reference manual, Ann Arbor, Mich., 1967.

〔Cris 65〕 Crisman, P. A., ED. The compatible time Sharing system: A programmer's guide, 2nd ed., M. I. T. Press, Combridge, Mass., 1965.

〔DAM 71〕 Van Dam, A. and Rice, D. E., On-line text editing: A survey, comput. surv. vol. 3, no3, pp. 93—114, September 1971.

〔Don 78〕 V. Donzeau-Gouge, G. Kahn, B. Lang, A complete machine-checked definition of a simple programming language using denotational semantics I. R. I. A. Research Report no. 330, October 1978.

〔Enge 73〕 Engelbart, D. C., Watson, R. W., and Norton, J. C. "The augmented Knowledge workship," in Proc. National Computer Conf., vol. 42, AFIPS Press, Arlington, Va., 1973, pp. 9—21.

〔FAJM 73〕 Fajman, R. "WYLBUR: An interactive text editing and remote job entry system" Comm. ACM 16, 5(May 1973), 314—322.

〔Hans 71〕 Hansen, W. J. "Creation of hierarchic text with a computer display," Rep. ANL 7818, Argonne National Laboratory, Argonne, Ill., July 1971.

〔IBM 67〕 International Business Machines Magnetic Tape selectric typewriter, New York, 1967.

〔Lamp 78〕 Lampson, B. W. "Bravo manual," in Alto user's handbook, Xerox Palo Alto Research Center, Palo Alto, Calif., Nov. 1978, pp. 31—62.

〔LED 80〕 Ledgard, H et al, The Natural Language of Interactive Systems, Comm. ACM, vol. 23, no. 1 O, pp. 556—563, October 1980.

〔SAN 78〕 Sandewall, E., Programming in an Interactive Environment: the Lisp Experience, ACM Comp. Surveys, vol. 10, no. 1, pp. 35—71, 1978.

〔Suth 63〕 Sutherland, I. E. "THOR-A display based timesharing system," in Proc. Spring Jt. Computer Conf., vol. 23, Spartan, Baltimore, 1963, p. 329.

〔TEI 78〕 Teitelman, W., The INTERLISP Reference Manual. Xerox PARC, Palo Alto, Calif, 1978.