SPECIFICATION OF A TOOL FOR VIEWING PROGRAM TEXT

Airi Salminen^{*}

ABSTRACT

The maintenance of large programs is a demanding process where lot of information is needed. Much of this information is in the program text. However, the finding of the needed information may be very difficult. It seems evident that more powerful tools are needed for helping the maintainers to find the information they need.

The paper introduces a method for extending information retrieval capabilities of document handling systems. The specific documents which are considered in this paper are program texts. For the specification we shall use notions of a generic document model where operations for selecting parts of documents as well as operations for manipulating documents have been defined. The model supports flexible and powerful retrieval capabilities from documents and it allows different views of the same documents. As an example we shall specify a tool for accessing information from Pascal programs.

*

University of Jyväskylä, Department of Computer Science
Seminaarinkatu 15, SF-40100, Jyväskylä, Finland

1. INTRODUCTION

A maintainer of a large program needs a lot of information concerning the program. Most of this information can be found in the program text. However, in case of large programs, the process of finding the appropriate parts - those that contain the needed information and not too much extra information - is not simple. There are several methods used for accessing information concerning programs: rummaging through the program listing, using retrieval capabilities of an editor or a program constructor, using a debugger and its capabilities for giving answers to some questions or making queries from some other kind of program database (e.g. [Archer and Conway], [Burkhart and Nievergelt], [Donzeau-Gouge et al.], [Habermann], [Horwitz and Teitelbaum], [Lewis], [Meyer et al.], [Shapiro et al.], [Smith et al.], [Sproull], [Strömfors and Jonesjö], [Warren] and [Wood]).

Considering the great difficulties appeared in the maintenance of programs and the great costs they cause [Leitntz and Swanson], it seems evident that more powerful tools are needed for helping the maintainer to find information about the program and to understand the program. However, it is not at all clear what kind of tool would be the most useful. Obviously different tools are needed for different environments and different users.

We will present a method for specifying tools for extending the document handling facilities of computer systems, particularly information retrieval from documents. The specific documents which are considered in this paper are program texts and the tool specified is intended to be used for helping a maintainer to find information from the program. As an example we shall specify one tool whose basic properties are the following: 1) The tool is used for accessing parts of a Pascal program text. 2) The accessed parts appear in a compacted program. 3) The compacted program can be further compacted or extended. 4) A cursor can be moved on the compacted program for showing the focus of interest. Other kind of tools, either for Pascal programs or for programs written in other languages (or for some other kind of documents), can be specified using the same method.

For the specification we shall use a general document model where operations for selecting parts of documents as well as operations for manipulating documents have been defined [Salminen and Back]. The document model is a generic model which can be regarded as a combination of the picture model of graphics [Mallgren] and the relational model of data [Codd]. We will consider the Pascal program and the compacted program as documents and we will define the commands for operating with these documents.

The definitions of the basic notions of the general document model are presented in Section 2. Section 3 describes in which way the document model is used as a specification method. The specification of the tool is presented in Sections 4 - 6. In Section 7 we show a sample session where the tool is used.

2. DOCUMENT MODEL

2.1 Documents

As the basic mathematical notion for defining properties of a set of objects we shall use many-sorted structures: A (many-sorted) structure $S = (SS; cc, rr, ff)$ consists of (i) a collection SS of nonempty sets (the carrier sets of S), (ii) a collection cc of constants c , where each constant $c \in cc$ is an element of some carrier set $S \in SS$, (iii) a collection rr of relations r , where each relation $r \in rr$ is some subset $r \subseteq S_1 \times \dots \times S_n$, $n > 0$ and $S_k \in SS$ for $k = 1, \dots, n$, and (iv) a collection ff of operations f , where each operation $f \in ff$ is some function $f: S_1 \times \dots \times S_n \rightarrow S_{n+1}$, $n > 0$ and $S_k \in SS$ for $k = 1, \dots, n+1$.

As a basic feature in all documents is regarded their duality: they have form (or layout) and content. Therefore we postulate two kinds of structures, a layout structure $\underline{L}$ and a content structure $\underline{C}$. A layout structure $\underline{L}$ contains a special carrier set L that will be referred to as the layout set. A content structure $\underline{C}$ contains a special carrier set C that will be referred to as the content set. A pair $T = (\underline{L}, \underline{C})$ is defined to be a document type. In a pair $T = (\underline{L}, \underline{C})$ the layout structure $\underline{L}$ describes what kind of form the documents of type T can have and the content structure $\underline{C}$ describes the properties of the data that can be put in the documents of type T .

A document of type $T = (\underline{L}, \underline{C})$ is a partial function $d: L \rightarrow C$, where L is the layout set of $\underline{L}$ and C is the content set of $\underline{C}$. The domain of a document, i.e. the subset L' of L for which d is defined, is called the layout of d and is denoted $\text{layout}(d)$. The range of a document d , i.e. the set of possible values of d , is called the content of d and is denoted $\text{content}(d)$. The document d associates with each element l in the layout of d a specific value $d(l)$ of C , a pair $(l, d(l)) \in d$ is a document element of d . The set of all documents of type T is denoted D_T . For all document types we define the empty document and a unit document. The empty document is denoted $\emptyset$ and its layout is the empty set. A unit document is denoted $\langle l, c \rangle$, where $l \in L$ and $c \in C$. It consists of one document element.

2.2 Document operations

We have identified four different types of operations that can be meaningfully defined for all documents: selecting a part of a document, combining documents,

changing the layout of a document and changing the content of a document. In the definitions we will use the following notations: $\cup$ for set union, $\cap$ for set intersection, $-$ for set difference and $;$ for function composition.

Selection operations

We can generate for all document types selection operations using a selection criterion. A selection criterion of a document type T is either a subset W_D of D_T , a subset W_L of the layout set L or a subset W_C of the content set C . In other words, a selection criterion is identified with a set of documents, with a set of layout elements or with a set of content elements. Each type of selection criterion determines a selection operation, a document selection $dselect(W_D): D_T \rightarrow D_T$, a layout selection $lselect(W_L): D_T \rightarrow D_T$ and a content selection $cselect(W_C): D_T \rightarrow D_T$, respectively. The operations are defined as

$$\begin{aligned} dselect(W_D)(d) &= \{(l, d(l)) \in d \mid (l, d(l)) \in w \text{ and } w \in W_D\}, \\ lselect(W_L)(d) &= \{(l, d(l)) \in d \mid l \in W_L\} \text{ and} \\ cselect(W_C)(d) &= \{(l, d(l)) \in d \mid d(l) \in W_C\}. \end{aligned}$$

A selection operation selects from any document d exactly that part of d that satisfies the selection criterion. Since the type of the selection operation can be seen from the type of the selection criterion, we can also use, whenever convenient, a common notation $select(W)$, where W is either W_D , W_L or W_C , for different selection operations. If the selection criterion is empty, then $select(W)(d) = \emptyset$ for all $d \in D_T$.

Combining operations

In a combining operation the layout of the resulting document is the union of the layouts of the operand documents. Since the layouts of the operand documents may overlap, we must define, what is the content of the resulting document in the common layout elements. Let $T = (\underline{L}, \underline{C})$ be a document type. A content combiner is a binary function $\psi: C \times C \rightarrow C$, where C is the content set of $\underline{C}$. A content combiner ψ determines a combine operation $combine(\psi): D_T \times D_T \rightarrow D_T$, defined by

$$combine(\psi)(d, d') = d''$$

where $layout(d'') = layout(d) \cup layout(d')$ and for each $l \in layout(d'')$

$$\begin{aligned} d''(l) &= d(l) \text{ , if } l \in layout(d) \text{ and } l \notin layout(d') \\ &= d'(l) \text{ , if } l \notin layout(d) \text{ and } l \in layout(d') \\ &= \psi(d(l), d'(l)) \text{ , if } l \in layout(d) \text{ and } l \in layout(d') \end{aligned}$$

Layout shifting operations

Layout shifting operations are used to change the layout of a document. A layout shift of a document type $T = (\underline{L}, \underline{C})$ is a partial function $\lambda: L \rightarrow L$, where L is the

layout set of $\underline{L}$. A layout shift λ determines an operation $\text{shift}(\lambda): D_T \rightarrow D_T$,
 $\text{shift}(\lambda)(d) = \lambda; d$.

Content changing operations

A content change is a partial function $\gamma: C \rightarrow C$, where C is the content set of the document type T . A content change γ determines an operation $\text{change}(\gamma): D_T \rightarrow D_T$,
 $\text{change}(\gamma)(d) = d; \gamma$.

2.3 Document structures

In a document structure we collect together all the information needed for handling documents of a given type. A document structure of type $T = (\underline{L}, \underline{C})$ is a structure $\underline{D}$ whose carrier sets contain the set D_T and where $\underline{L}$ and $\underline{C}$ appear as substructures.

In a document structure specific operations on documents can be generated from generic document operations using properties of $\underline{L}$ and $\underline{C}$ and relations and constants of $\underline{D}$. Partial functions used as layout shifts or content changes may be defined directly or they may be defined with functions which produce from some arguments a layout shift or a content change. We call a function $\tau: S_1 \times \dots \times S_k \rightarrow L_T$, where $k > 0$, S_i is a carrier set of $\underline{D}$ for $i=1, \dots, k$ and L_T is the set of all layout shifts of T , a layout transformer. Respectively we call a function $\rho: S_1 \times \dots \times S_k \rightarrow C_T$, where C_T is the set of all content changes of T , a content transformer.

2.4 Document databases

A document structure $\underline{D}$ of type T defines a calculus for handling documents of type T . For describing the way how this calculus is used we can define a new document structure $\hat{\underline{D}}$ which we call a document database of $\underline{D}$. In it we define the components of a state of the document database, the possible states of the database, the legal commands of the database and their semantics. A state of a document database is itself a document consisting of the components of the state and their values. (More generally a document database can be defined for a collection of document structures.)

Let $\underline{D}$ be a document structure, $t_1, \dots, t_n$ ($n > 0$) some names, $S_1, \dots, S_n$ and $P_1, \dots, P_k$ ($k > 0$) some carrier sets of $\underline{D}$ and $\hat{C}$ a subset of $S_1 \times \dots \times S_n$. Then a document database of $\underline{D}$ is a document structure $\hat{\underline{D}}$ of type $\hat{T} = (\hat{\underline{L}}, \hat{\underline{C}})$ where $\hat{\underline{L}} = (\{(t_1, \dots, t_n)\}; \emptyset, \emptyset, \emptyset)$ and $\hat{\underline{C}} = (\{\hat{C}, P_1, \dots, P_k\}; \emptyset, K, \emptyset)$. The only carrier set of $\hat{\underline{L}}$ is also the layout set, the content set of $\hat{\underline{C}}$ is $\hat{C}$. The relations of K are content change functions and content transformers and $\hat{\underline{D}}$ has the following properties: a) $\underline{D}$ appears as its substructure, b) one of its carrier sets is the set K' of all content change functions defined in K either directly or by a content transformer, c) one of its

constants is a document of $D_{\hat{T}}$ and d) one of its operations is the operation
 changestate: $K' \times D_{\hat{T}} \rightarrow D_{\hat{T}}$ defined as

$$\text{changestate}(k,s) = \text{change}(k)(s)$$

for all $k \in K'$ and $s \in D_{\hat{T}}$. We call the names $t_1, \dots, t_n$ the components of $\hat{D}$, the set $\hat{C}$ the states of $\hat{D}$, the sets $P_1, \dots, P_k$ the legal parameters of $\hat{D}$, the functions of K the legal commands of $\hat{D}$ and the constant document of type $\hat{T}$ the initial state of $\hat{D}$.

Above we defined a legal command of a document database to be either a content change function or a content transformer function. In this way we need not to be concerned about the syntax of the commands. In different environments commands may be given in different ways, e.g. by selecting from a menu, by using function keys or by writing them explicitly using some specific syntax.

3. THE DOCUMENT MODEL AS A SPECIFICATION METHOD

A document database defines, at some convenient level of abstraction, a tool for handling documents. The selected abstraction level depends on the environment where the tool is intended to be used. For example, if the tool will be an extension of a system where window manipulation operations are existing, there is no need for describing window manipulation operations in the document database. On the other hand, if we do not have suitable window handling mechanism, we can define it as a part of the document database. The specification can also be more or less formal, depending on the selected specification language. If we give the specification using a programming language (e.g. Prolog), the specification is also a prototype of the tool. The specification consists of three parts: first we specify the document type T , then the document structure D of type T and finally, the document database of D .

For our sample specification we extend our specification language with relational operations intersection ($\wedge$), difference ($-$), selection (σ_F), projection (π_i) and restriction ($_{(i)}|_{(j)}$) [Codd]. The formula F in the selection operation is given in the form $i=G$ where $i>0$ and G is an element on the i 'th domain of the relation. The operation $R_{(i)}|_{(j)}S$ selects from the relation R those tuples where the i 'th component is equal to the j 'th component of some tuple of relation S . We will also take the properties of integers as well as the notions of Pascal language for granted. The document type of Pascal programs is described in Section 4, the document structure for Pascal programs in Section 5 and the document database for handling Pascal programs in Section 6.

4. THE DOCUMENT TYPE TEXT

The documents of type *TEXT* consist of characters locating on line *i* on position *j*.

Layout structure

structure *POSITIONS*

carriersets: $N \times N$, N

N is the set of natural numbers, $N \times N$ is the layout set.

Content structure

structure *CHARACTERS*

carriersets: B

operations: *covering*: $B \times B \rightarrow B$

B is the character set of Pascal programs and B is also the content set. The function *covering* will be used as a content combiner and it is defined as

$$\text{covering}(x, y) = y.$$

5. THE DOCUMENT STRUCTURE PASCALTEXT

We define here a document structure of type *TEXT* which specifies a calculus for handling parts of a Pascal program. The relations of the document structure will be used for expressing selection criteria when selecting parts of the program. Therefore we collect to the relations all those properties of parts of the program which we want to use as selection criteria.

structure *PASCALTEXT*

substructures: *POSITIONS*, *CHARACTERS*

carriersets: D_{TEXT} , M , R

constants: *p*, *linenumbers*, *programheading* $\in D_{\text{TEXT}}$; *i0*, *j0* $\in N$

relations: *DECLARATION* $\subseteq M \times D_{\text{TEXT}}$

SUBROUTINE $\subseteq D_{\text{TEXT}}$

DECLARATION-PART, *STATEMENT-PART*, *BRACKETS*,

LOOP, *IF*, *CASE*, *WITH*, *READ*, *WRITE*,

NEXTBRACKETS, *USES*, *PARTOF*, *SOMEPARTIN* $\subseteq D_{\text{TEXT}} \times D_{\text{TEXT}}$

operations: *firstline*: $D_{\text{TEXT}} \rightarrow N \times N$

up, *down*, *left*, *right*: $D_{\text{TEXT}} \times (N \times N) \rightarrow N \times N$

$\oplus$, $\otimes$, $\odot$, *select-lines*: $D_{\text{TEXT}} \times D_{\text{TEXT}} \rightarrow D_{\text{TEXT}}$

show-where, *next-brackets*, *outermost-brackets*, *innermost-brackets*:

select-char: $N \times N \rightarrow D_{\text{TEXT}}$ $D_{\text{TEXT}} \rightarrow D_{\text{TEXT}}$

select-subroutine, *select-all-decl*: $M \rightarrow D_{\text{TEXT}}$

select-decl: $D_{\text{TEXT}} \rightarrow D_{\text{TEXT}}$

select-uses: $M \times D_{\text{TEXT}} \rightarrow D_{\text{TEXT}}$

select-part: $R \times M \rightarrow D_{\text{TEXT}}$

Carrier sets

M is the set of Pascal identifiers and R is the set $\{ \text{DECLARATION-PART, STATEMENT-PART, LOOP, IF, CASE, WITH, READ, WRITE} \}$ of relation names.

Constants

The constant document p is the Pascal program from where we are retrieving information. The constant document $linenumbers$ contains all line numbers of p and it is used for showing to the user (when wanted) the line numbers of the lines accessed to the compacted program. The line numbers are shown in the beginning of each line. The constant $j_0 \geq 0$ gives the number of character positions reserved to the line numbers on lines. The documents p and $linenumbers$ are defined so that $layout(p) = \{(i,j) \mid 1 \leq i \leq \text{maxlinenumber}, j_0 < j \leq n_1\}$ where maxlinenumber is the number of lines in the program and $n_1 > j_0$, and $layout(linenumbers) = \{(i,j) \mid 1 \leq i \leq \text{maxlinenumber}, 1 \leq j \leq j_0\}$. The constant document $programheading$ contains the heading line(s) of p . This document will always be at the compacted program. The constant i_0 is the line number of the first line of $programheading$.

Relations

In a pair (m,d) of the relation $DECLARATION$ m is an identifier declared or defined in p on line(s) d . The unary relation $SUBROUTINE$ contains all procedure and function declarations of p as well as the whole p . From now on the documents of $DECLARATION$ are called declarations and the documents of $SUBROUTINE$ are called subroutines. The presence of a tuple (d,d') in relations $DECLARATION$ -PART and $STATEMENT$ -PART means that d' is the declaration part or the statement part of the subroutine d , respectively. In a (d,d') of $BRACKETS$ relation, d is a subroutine and d' consists of the heading and ending lines of the subroutine. In tuples (d,d') of $LOOP$, IF , $CASE$, $WITH$, $READ$ and $WRITE$ d is again a subroutine and d' consists of such lines in the statement part of the subroutine where a statement of the specific type appears. The documents in the binary relation $NEXTBRACKETS$ appear as second components in tuples of $BRACKETS$. If (d,d') is a tuple of $NEXTBRACKETS$ then there are tuples (s,d) and (s',d') in $BRACKETS$ such that $s' \subseteq s$ and there is no such tuple (s'',d'') in $BRACKETS$ relation that $s'' \subseteq s$ and $s' \subseteq s''$. All documents of the above relations consist of whole lines of p .

The relation $USES$ contains a pair (d,d') for each use d' of an identifier declared in p , d is a declaration. In pairs (d,d') of $PARTOF$ $d \subseteq d'$ and in pairs (d,d') of $SOMEPARTIN$ at least one document element of d is also a document element of d' . Relations $PARTOF$ and $SOMEPARTIN$ are defined for all nonempty subsets of p .

Operations

Operations *firstline*, *up*, *down*, *left* and *right* all yield a layout element of a given document. We define operations *firstline*, *up* and *left*.

$$\begin{aligned}
firstline(d) &= (i0, j0+1) && \text{if } d = \emptyset \text{ and otherwise} \\
&= (i', j0+1) && \text{where } i' = \min\{i \mid (i, j) \in layout(d)\} \\
up(d, (n, m)) &= (i0, j0+1) && \text{if } (n, m) \notin layout(d), \\
&= (n, m) && \text{if } n = \min\{i \mid (i, j) \in layout(d)\} \text{ and otherwise} \\
&= (n', m) && \text{where } n' = \max\{i \mid (i, j) \in layout(d) \text{ and } i < n\} \\
left(d, (n, m)) &= (i0, j0+1) && \text{if } (n, m) \notin layout(d), \\
&= (n, j0+1) && \text{if } m \leq j0+1 \text{ and otherwise} \\
&= (n, m') && \text{where } m' = \max\{j \mid (i, j) \in layout(d) \text{ and } j < m\}
\end{aligned}$$

Operations $\oplus$, $\otimes$, $\odot$ and *select-lines* construct from two documents of type *TEXT* a new document of type *TEXT*. They are defined in the following way:

$$\begin{aligned}
d1 \oplus d2 &= combine(covering)(d1, d2) \\
d1 \otimes d2 &= lselect(layout(d1) - layout(d2))(d1) \\
d1 \odot d2 &= lselect(layout(d1) \wedge layout(d2))(d1) \\
select-lines(d1, d2) &= lselect(\{(i, j) \in layout(d1) \mid (i, k) \in layout(d2) \\
&\quad \text{for some } k\})(d1)
\end{aligned}$$

The rest of the operations select a part from the constant document p . The operations *show-where*, *next-brackets*, *outermost-brackets* and *innermost-brackets* result either the empty document or a document consisting of heading and ending lines of subroutines. They are defined as

$$\begin{aligned}
show-where(d) &= select(\pi_2(BRACKETS_{(1)} \mid (2) \sigma_{1=d}(SOMEPARTIN)))(p) \\
next-brackets(d) &= select(\pi_2((NEXTBRACKETS_{(1)} \mid (1) \sigma_{2=d}(PARTOF)) - \\
&\quad (NEXTBRACKETS_{(2)} \mid (1) \sigma_{2=d}(PARTOF))))(p) \\
outermost-brackets(d) &= select(\pi_1(NEXTBRACKETS_{(1)} \mid (1) \sigma_{2=d}(PARTOF)) - \\
&\quad \pi_2(NEXTBRACKETS_{(2)} \mid (1) \sigma_{2=d}(PARTOF)))(p) \\
innermost-brackets(d) &= select(\pi_2(NEXTBRACKETS_{(2)} \mid (1) \sigma_{2=d}(PARTOF)) - \\
&\quad \pi_1(NEXTBRACKETS_{(1)} \mid (1) \sigma_{2=d}(PARTOF)))(p)
\end{aligned}$$

The operation *select-char* selects from p one document element and *select-subroutine* selects all subroutines which have a given name.

$$\begin{aligned}
select-char(n, m) &= \emptyset && \text{if } (n, m) \notin layout(p) \text{ and otherwise} \\
&= lselect(\{(n, m)\})(p) \\
select-subroutine(m) &= select(\pi_2(\sigma_{1=m}(DECLARATION)) \wedge SUBROUTINE)(p)
\end{aligned}$$

For selecting a declaration we have two operations. The argument of *select-all-decl* is an identifier and this operation selects from p all those lines where some declaration of the identifier appears. In the operation *select-decl* again the argument is a document. If this document is some part of a name used in p , then the result of the operation is the declaration of the name. Otherwise the resulting document is empty. The operation *select-uses* has two arguments, an identifier and a document.

If the document is a part of a declaration and the identifier is declared in that declaration, then the resulting document consists of such lines of p where the identifier appears representing the declared entity. Finally, the operation *select-part* selects a given part from the subroutines which have a given name. These operations are defined as

$$\begin{aligned}
 \text{select-all-decl}(m) &= \text{select}(\pi_2(\sigma_{1=m}(\text{DECLARATION}))) (p) \\
 \text{select-decl}(d) &= \text{select}(\pi_1((\text{USES}_{(2)} \mid_{(2)} \sigma_{1=d}(\text{PARTOF})))) (p) \\
 \text{select-uses}(m,d) &= \text{select-lines} \\
 &\quad (p, (\text{select}(\pi_2((\text{USES}_{(1)} \mid_{(2)} \sigma_{1=d}(\text{PARTOF}))_{(1)} \mid_{(2)} \sigma_{1=m}(\text{DECLARATION})))) (p))) \\
 \text{select-part}(r,m) &= \text{select}(\pi_2(r_{(1)} \mid_{(2)} \sigma_{1=m}(\text{DECLARATION}))) (p)
 \end{aligned}$$

6. THE DOCUMENT DATABASE FOR PASCAL PROGRAMS

We are specifying a tool for accessing parts of the program p to the compacted program and we want to move cursor in the compacted program for showing the focus of our interest. We define the tool as a document database of PASCALTEXT in the following way.

components: *compacted, cursor*

states: $\hat{C} = \{(co, cu) \mid co \in D_{\text{TEXT}}, co \subseteq p \cup \text{linenumbers}, \text{programheading} \subseteq co \text{ and } cu \in \text{layout}(co) \cap \text{layout}(p)\}$

initial state: $\{((\text{compacted}, \text{cursor}), (\text{programheading}, (i0, j0)))\}$

legal parameters: M, R

legal commands: MOVE CURSOR UP, MOVE CURSOR DOWN, MOVE CURSOR RIGHT, MOVE CURSOR LEFT, SHOW STRUCTURE, SHOW OUTERMOST STRUCTURE, ADJUST STRUCTURE, COMPACT STRUCTURE, WHERE, WHERE THIS LINE, SHOW DECLARATION, SHOW LINENUMBERS, HIDE LINENUMBERS, START NEW: $\hat{C} \rightarrow \hat{C}$

SHOW DECLARATION OF, SHOW USES OF, SHOW STRUCTURE OF, SHOW OUTERMOST STRUCTURE OF, ADJUST STRUCTURE OF, COMPACT STRUCTURE OF : $M \rightarrow C_{\hat{T}}$

SHOW PART OF : $R \times M \rightarrow C_{\hat{T}}$

The first group of the commands consists of content change functions, the rest of the commands are content transformers. We define the commands as follows.

$$\begin{aligned}
 \text{MOVE CURSOR UP } (co, cu) &= (co, up(co, cu)) \\
 \text{MOVE CURSOR DOWN } (co, cu) &= (co, down(co, cu)) \\
 \text{MOVE CURSOR RIGHT } (co, cu) &= (co, right(co, cu)) \\
 \text{MOVE CURSOR LEFT } (co, cu) &= (co, left(co, cu)) \\
 \text{SHOW STRUCTURE } (co, cu) &= (co @ show-where(p), cu) \\
 \text{SHOW OUTERMOST STRUCTURE } (co, cu) &= (co @ next-brackets(outermost-brackets(p)), cu) \\
 \text{ADJUST STRUCTURE } (co, cu) &= (co @ next-brackets(co), cu)
 \end{aligned}$$

```

COMPACT STRUCTURE (co,cu) = (co⊗select-lines(co, innermost-brackets(co)), cu)
WHERE (co,cu) = (co⊗show-where(select-lines(p,co)), cu)
WHERE THIS LINE (co,cu) = (co⊗show-where(select-char(cu)), cu)
SHOW DECLARATION (co,cu) = (co⊗select-decl(select-char(cu)),
                             firstline(select-decl(select-char(cu))))
SHOW LINENUMBERS (co,cu) = (co⊗select-lines(linenumbers,co), cu)
HIDE LINENUMBERS (co,cu) = (co⊗linenumbers, cu)
START NEW (co,cu) = (programheading, (i0,j0))
SHOW DECLARATION OF (m) (co,cu) = (co⊗select-all-decl(m),
                                     firstline(select-all-decl(m)))
SHOW USES OF (m) (co,cu) = (co⊗select-uses(m,select-char(cu)),
                             firstline(select-uses(m, select-char(cu))))
SHOW STRUCTURE OF (m) (co,cu) = (co⊗show-where(select-subroutine(m)), cu)
SHOW OUTERMOST STRUCTURE OF (m) (co,cu) =
    (co⊗next-brackets(outermost-brackets(select-subroutine(m)),cu))
ADJUST STRUCTURE OF (m) (co,cu) =
    (co⊗next-brackets(select-subroutine(m)⊗co), cu)
COMPACT STRUCTURE OF (m) (co,cu) =
    (co⊗select-lines(co,innermost-brackets(select-subroutine(m)⊗co)), cu)
SHOW PART (r) OF (m) (co,cu) = (co⊗select-part(r,m),
                                  firstline(select-part(r,m)))

```

7. A SAMPLE SESSION

In a session we might have e.g. the following commands:

- (1) SHOW STRUCTURE OF scanner
- (2) SHOW STATEMENT-PART OF scantoken
- (3) MOVE CURSOR DOWN
- (4) MOVE CURSOR RIGHT
- (5) SHOW DECLARATION
- (6) START NEW
- (7) SHOW DECLARATION OF token
- (8) WHERE
- (9) MOVE CURSOR DOWN
- (10) SHOW USES OF token
- (11) SHOW LINENUMBERS

The first command accesses the structure of the procedure 'scanner' (i.e. the beginning and ending lines of the procedure and function declarations appearing in 'scanner'). From the compacted program we see that one of the procedures inside 'scanner' is 'scantoken'. The second command appends the statement part of this procedure to the compacted program. The commands (3) and (4) are performed several times, until the cursor points a name. The fifth command retrieves the declaration

of the pointed name. By the START NEW command the previously accessed information is removed from the compacted program. The command (7) accesses all declarations of the name 'token'. Suppose 'token' is declared twice and we want to see, where the declarations appear in the program. The command (8) shows us the beginning and ending lines of those procedures and functions inside which the declarations are locating. At this stage we are interested in the last declared 'token'. Therefore we move cursor down to the second declaration. The command (10) retrieves to the compacted program all lines where this 'token' is used. Finally, the last command shows the line numbers of the lines of the program which we have accessed.

Acknowledgements: I wish to thank Ralph-Johan Back for many useful discussions and for reading and commenting on earlier drafts of this paper.

REFERENCES

1. Archer, J. and Conway, R., Display condensation of program text, IEEE Trans. Softw.Eng. 8,5 (Sep. 1982), 526-529.
2. Burkhart, H. and Nievergelt, J., Structure-oriented editors, Institut für Informatik ETH, Zürich, 1980.
3. Codd, E.F., A relational model of data for large shared data banks, Commun.ACM 13,6 (June 1970), 377-387.
4. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, G., Programming environments based on structure editors: the MENTOR experience, IRIA, 1980.
5. Habermann, A.N., System development environments, in Tools and Notions for Program Construction (ed. by D. Néel), Cambridge Univ. Press, Cambridge, 1982.
6. Horwitz, S., Teitelbaum, T., Relations and attributes: a symbiotic basis for the editing environments, ACM SIGPLAN Not. 20,7 (July 1985), 93-106.
7. Leintz, B. and Swanson, E.B., Software Maintenance Management, Addison-Wesley, 1980.
8. Lewis, J.W., Beyond ALBE/P: Language Neutral Form. 5th Int. Conf. on Software Engineering, IEEE Computer Society (1981), 422-429.
9. Mallgren, W.R., Formal Specification of Interactive Graphics Programming Languages, The MIT Press, 1985.
10. Meyer, B., Nerson, J.-M., Ko, S.H., Showing programs on a screen, Science of Computer Programming 5,2 (June 1985), 111-142.
11. Salminen, A. and Back, R.J.R., A relational model for documents, Proc. of the Int. Symposium on New Directions in Computing, IEEE Computer Society (1985), 50-59.
12. Shapiro, D.G., Dean, J.S., McCune, B.P., A knowledge base for supporting an intelligent program editor, 7th Int. Conf. on Software Engineering, IEEE Computer Society (1984), 381-386.
13. Smith, S.R., Barnard, D.T., Macleod, I.A., Holophrasted displays in an interactive environment, Int.J.Man-Mach.Stud. 20,4 (April 1984), 343-355.
14. Sproull, R.F., Raster graphics for interactive programming environments, ACM SIGGRAPH 13,2 (August 1979), 83-93.
15. Strömfors, O. and Jonesjö, L., The implementation and experiences of a structure oriented text editor, ACM SIGPLAN Not. 16,6 (June 1981), 22-27.
16. Warren, S., MAP: a tool for understanding software, Proc. of 6th Int. Conf. on Software Engineering (1982), 28-37.
17. Wood, S.R., Z - The 95 % program editor, ACM SIGPLAN Not. 16,6 (June 1981), 1-7.