A Structured Method for Literate Programming

by

Sho-Huan Tung

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 269

A Structured Method for Literate Programming

by

Sho-Huan Tung

January, 1989

Research reported herein was supported in part by the National Science Foundation, under grant number CCR 88-03432.

To appear in *Structured Programming: An International Journal.*

A Structured Method for Literate Programming

Sho-Huan Tung

Computer Science Department
Indiana University
Bloomington, Indiana 47405

Abstract

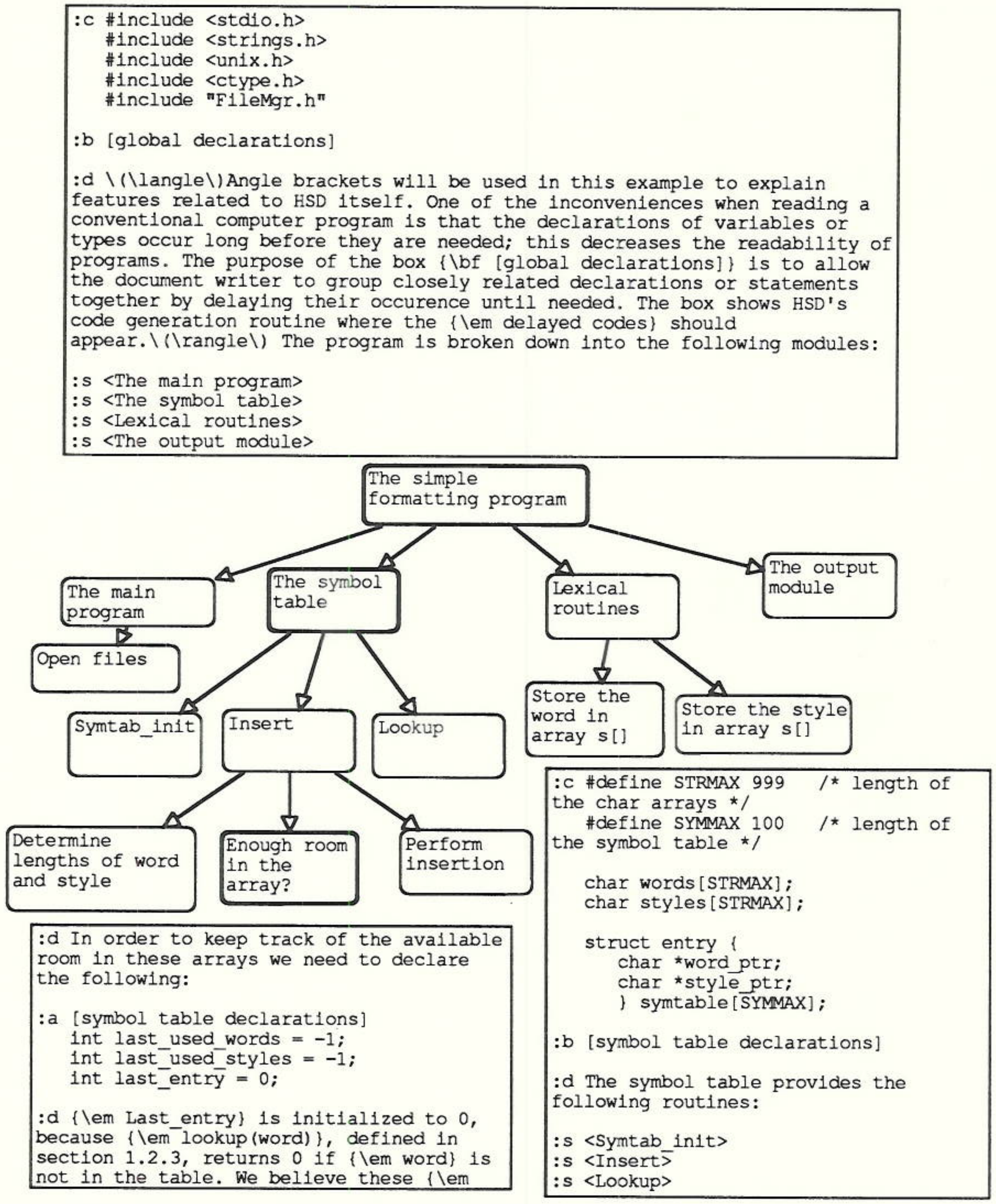
In order to make computer programs easier to comprehend, the methods for program development and documentation need to be enhanced beyond their conventional treatment. Literate programming is an approach to programming that emphasizes improving the readability of computer programs. This paper describes a structured method for literate programming. HSD (Hierarchical Structured Document) is a tool that supports automatic code and document generation from a hierarchically structured document description. An example is given to illustrate the use of HSD. The way in which empirical studies on programmers' behavior influence the design of HSD and some directions for future research are addressed.

A. Introduction

The readability of computer programs is an important concern in software development. More readable programs are easier to understand and maintain, thus reducing the cost of software development. Much progress has been made in improving the readability of computer programs. Structured design methodologies have improved the logical structure of programs. The advanced development of programming language constructs, such as modules, objects, and procedural abstractions, have improved the clarity of programs. The use of digital typography techniques can improve the appearance of programs. Despite these improvements, the readability of many computer programs is still not very satisfactory. The basic problem is that most programs are *not* written for humans to read; they are written for computers to execute.

Knuth proposes an approach to programming called literate programming [9] to alleviate the problem. Literate programs are written for humans to read as well as for computers to execute. He developed the WEB programming system to allow programmers to write literate programs. The philosophy behind the WEB system is that a program is mentally represented as a web of ideas. The challenge to a WEB programmer is to transform the web of ideas into an order that best illustrates the program. The WEB system takes care of the details of generating the documentation and the program from the specification of the ideas. I believe Knuth's approach is very promising. However, the major drawback of the WEB system is that the document produced with the WEB is organized linearly, thus it does not capture the structure of the ideas explicitly.

The primary reason literate programs are more readable than their traditional counterparts is that literate programs are presented in a manner that match better with programmers' mental representation of programs. This paper presents a programming tool which supports a structured


The simple formatting program
The main program
Open files
The symbol table
Symtab_init
Insert
Lookup
Determine lengths of word and style
Enough room in the array?
Perform insertion
Lexical routines
Store the word in array s[]
Store the style in array s[]
The output module

```
:c #include <stdio.h>
   #include <strings.h>
   #include <unix.h>
   #include <ctype.h>
   #include "FileMgr.h"

:b [global declarations]

:d \(\langle\)Angle brackets will be used in this example to explain
features related to HSD itself. One of the inconveniences when reading a
conventional computer program is that the declarations of variables or
types occur long before they are needed; this decreases the readability of
programs. The purpose of the box {\bf [global declarations]} is to allow
the document writer to group closely related declarations or statements
together by delaying their occurence until needed. The box shows HSD's
code generation routine where the {\em delayed codes} should
appear.\(\rangle\) The program is broken down into the following modules:

:s <The main program>
:s <The symbol table>
:s <Lexical routines>
:s <The output module>
```

```
:c #define STRMAX 999   /* length of
the char arrays */
   #define SYMMAX 100   /* length of
the symbol table */

   char words[STRMAX];
   char styles[STRMAX];

   struct entry {
      char *word_ptr;
      char *style_ptr;
      } symtable[SYMMAX];

:b [symbol table declarations]

:d The symbol table provides the
following routines:

:s <Symtab_init>
:s <Insert>
:s <Lookup>
```

```
:d In order to keep track of the available
room in these arrays we need to declare
the following:

:a [symbol table declarations]
   int last_used_words = -1;
   int last_used_styles = -1;
   int last_entry = 0;

:d {\em Last_entry} is initialized to 0,
because {\em lookup(word)}, defined in
section 1.2.3, returns 0 if {\em word} is
not in the table. We believe these {\em
```

Fig. 1

method for literate programming. Unlike the WEB system which adopts the philosophy that a program is mentally represented as a web of ideas, HSD encourages programmers to weave the web into a structured representation such as a tree. This is the biggest difference between HSD and WEB.

An HSD user employs both graphical objects and text to write a literate program. (See Fig. 1.) Graphical objects are used to express the structure of the program. Text is used to provide detailed explanation of the program. The combination of graphical objects and text forms the specification of a literate program. This specification can be used to generate a program for computers to execute and documentation for humans to read.

In the next section, I present an example document developed with HSD. In Section C, I discuss the details of HSD's graphical document descriptive language and its user interface. I then discuss related ideas which have influenced the design of HSD. I conclude this paper by pointing out some possible directions for future research.

B. An Example Document

Consider a program that reads a text file, which I will call *fin*, and produces another text file *fout* that is just like *fin* except that all the occurences of *begin* in *fin* are replaced with {*bf begin*} and all the occurences of *end* in *fin* are replaced with {*bf end*}. Let's write a simple formatting program that functions just like the one described above except that the *begin*/{*bf begin*}, and *end*/{*bf end*} pairs or other additional *word/style* pairs are given in another input file designated *fword_style*. This program, with some improvements, is useful for formatting computer source codes in a language-independent manner. For example, all the keywords of a computer program can be formatted in bold-face (*bf* is the LaTeX command for bold-face[10]). The simple formatting program is written in HSD's graphical document descriptive language with LightspeedC[1] as the intended programming system. The following documentation of the simple formatting program is derived from the specification presented in Fig. 1. Note that the table of contents is hierarchically structured.

Table of contents:

[1]LightspeedC is a trade mark of Think Technology Inc.

1 The simple formatting program

This document describes the simple formatting program specified previously. Here are the "include" files needed in our solution.

```
#include <stdio.h>
#include <strings.h>
#include <unix.h>
#include <ctype.h>
#include "FileMgr.h"
```

Define box: **[global declarations]**

⟨Angle brackets will be used in this example to explain features related to HSD itself. One of the inconveniences when reading a conventional computer program is that the declarations of variables or types occur long before they are needed; this decreases the readability of programs. The purpose of the box **[global declarations]** is to allow the document writer to group closely related declarations or statements together by delaying their occurence until needed. The box shows HSD's code generation routine where the *delayed codes* should appear.⟩ The program is broken down into the following modules:

<The main program>
<The symbol table>
<Lexical routines>
<The output module>

⟨An HSD program is composed of a hierarchically ordered collection of sections. This section is in the highest level of the hierarchy, and its serial number is 1. The notation **<The main program>** means the following: The program text to be inserted here is called **The main program** and it is specified as a subsection of section 1.⟩

1.1 The main program

The main program opens files, initializes the symbol table, and then produces the output file by converting the input file according to the formats stored in the symbol table. The parameters to the main program are used to allow the user to specify the arguments (i.e. file names) of the program.

```
_main(argc, argv)
int argc;
char *argv[];
{
    if (argc != 4) {
        printf("format error: wrong number of arguments\n");
```

```
            exit();
            }
        else {
            <Open files>
            symtab_init();
            produce_output();
            exit();
            }
    }
```

1.1.1 Open files

```
    if ((fword_style = fopen(*++argv,"r")) == NULL) {
        printf("format error: can't open %s\n", *argv);
        exit();
        };
    if ((fin = fopen(*++argv, "r")) == NULL) {
        printf("format error: can't open %s\n", *argv);
        exit();
        };
    if ((fout = fopen(*++argv, "w")) == NULL) {
        printf("format error: can't open %s\n", *argv);
        exit();
        };
```

⟨After establishing the context for the occurence of the following variables, we can append them to the place where they syntactically belong.⟩

```
  Add code to: [global declarations]
   FILE *fword_style, *fin, *fout;
```

⟨A section in an HSD program can have comments, codes, names of subsections, as well as the commands *define-box* and *add-code-to-box*. They can occure optionally and in arbitrary order.⟩

1.2 The symbol table

The purpose of the symbol table is to provide a data structure to store the word-style pairs so that the rest of the program can generate output based on this information. Let us choose to use two char arrays *words[STRMAX]* and *styles[STRMAX]* to store the words and styles in the word-style file. In order to simplify this example, we will use another array *symtable[SYMMAX]* to store pointers which point to the positions where *word* and *style* are stored. The *symtable[SYMMAX]* array would normally be replaced by a hash table in a more efficient implementation.

```
    #define STRMAX 999 /* length of the char arrays */
```

```
#define SYMMAX 100 /* length of the symbol table */

char words[STRMAX];
char styles[STRMAX];

struct entry {
     char *word_ptr;
     char *style_ptr;
     } symtable[SYMMAX];
```

Define box: **[symbol table declarations]**

The symbol table provides the following routines:

<Symtab_init>
<Insert>
<Lookup>

1.2.1 Symtab_init

The codes in this section initialize the symbol table by using *get_word*, *get_style*, and *insert*. The functions *get_word* and *get_style* each take two parameters: a pointer to a character array, and a FILE pointer; they store the word or style in character arrays (*temp_word[]* and *temp_style[]*) and return 1 if a word or style is found and return 0 otherwise. Function *get_word* and *get_style* are defined in Section 1.3, "Lexical routines."

Add code to: **[symbol table declarations]**

```
#define BSIZE 128
char temp_word[BSIZE];
char temp_style[BSIZE];

symtab_init()
{
     int c;
     int b = 0;
     int more;
     more = get_word(temp_word, fword_style);
     while (more) {
          more = get_style(temp_style, fword_style);
          if (more)
               insert(temp_word, temp_style);
          else {
               printf("format error: word_style file not in pair\n");
               exit();
               }
          more = get_word(temp_word, fword_style);
          }
}
```

1.2.2 Insert

The function *insert(word, style)* puts *word* and *style* into *words[]* and *styles[]*, and puts the pointers, which point to *word* and *style*, in *symtable[]*.

```
int insert(word, style)
     char word[];
     char style[];
{
     <Determine lengths of word and style>
     <Enough room in the array?>
     <Perform insertion>
}
```

1.2.2.1 Determine lengths of word and style

```
int len_word;
int len_style;
len_word = strlen(word); /* library function for length of string */
len_style = strlen(style);
```

1.2.2.2 Enough room in the array?

In order to keep track of the available room in these arrays we need to declare the following:

```
Add code to: [symbol table declarations]
 int last_used_words = -1;
 int last_used_styles = -1;
 int last_entry = 0;
```

Last_entry is initialized to 0, because *lookup(word)*, defined in section 1.2.3, returns 0 if *word* is not in the table. I believe these *delayed declarations* improve the readability of the document, because if these declarations were declared in Section 1.2, the reader could be overwhelmed with too many declarations. The WEB system has a similar mechanism.

```
if (last_entry + 1 >= SYMMAX) {
     printf("format error: symbol table full");
     exit();
     }
if ((last_used_words + len_word + 1>= STRMAX) ||
     (last_used_styles + len_style + 1>= STRMAX)) {
     printf("format error: word/style array full");
     exit();
     }
```

1.2.2.3 Perform insertion

```
last_entry = last_entry + 1;
symtable[last_entry].word_ptr = &words[last_used_words + 1];
symtable[last_entry].style_ptr = &styles[last_used_styles + 1];
last_used_words = last_used_words + len_word +1;
last_used_styles = last_used_styles + len_style + 1;
strcpy(symtable[last_entry].word_ptr, word); /* string copy */
strcpy(symtable[last_entry].style_ptr, style);
return last_entry;
```

1.2.3 Lookup

The function *lookup(word)* looks up *word* in the *symtable*, returns the index of the entry if found, and returns 0 otherwise. The function *strcmp* is a library function which compares two strings.

```
int lookup(word)
    char word[];
{
    int p;
    for (p = last_entry; p > 0; p = p - 1)
        if (strcmp(symtable[p].word_ptr, word)== 0)
            return p;
    return 0;
}
```

1.3 Lexical routines

Two functions are defined in this section: The function *get_word(s, f)* reads from FILE *f*, stores the word in *s[]*, and returns 1. If no word is found, *get_word(s,f)* returns 0. The *get_style(s, f)* performs similarly. A *word* in the *word/style* file starts with a letter and is followed by letters, numbers, or the symbol '_'. A *style* starts with any characters except the white space characters and continues until a newline or EOF character is found. The word or style which is stored in *s[]* always ends with the character '\0' to form a character string in C.

```
int get_word(s, f)
    char s[];
    FILE *f;
{
    int c;
    int b = 0;
    while(1) {
        c = fgetc(f);
        if (c == EOF)
            return 0;
```

```
          else if (! isalpha(c))
               ;
          else {
               <Store the word in array s[]>
               }
          }
}
int get_style(s, f)
     char s[];
     FILE *f;
{
     int c;
     int b = 0;
     while(1) {
          c = fgetc(f);
          if (c == EOF)
             return 0;
          else if (c == ' ' || c == '\t')
               ;
          else if (c != '\n') {
               <Store the style in array s[]>
               }
          }
}
```

1.3.1 Store the word in array s[]

```
while (isalnum(c) || c == '_') {
     s[b] = c;
     c = fgetc(f);
     b = b + 1;
     if (b >= BSIZE) {
          printf("format error: buffer full in get_word\n");
          exit();
          }
     }
s[b] = '\0';
if (c != EOF)
     ungetc(c, f);
return 1;
```

1.3.2 Store the style in array s[]

```
while ((c != '\n') && (c !=EOF)) {
     s[b] = c;
     c = fgetc(f);
     b = b + 1;
     if (b >= BSIZE) {
          printf("format error: buffer full in get_style\n");
          exit();
          }
     }
s[b] = '\0';
return 1;
```

1.4 The output module

The function *produce_output()* reads the file *fin*, echoes to the file *fout* with *word* replaced by *style* for those words defined in the symbol table.

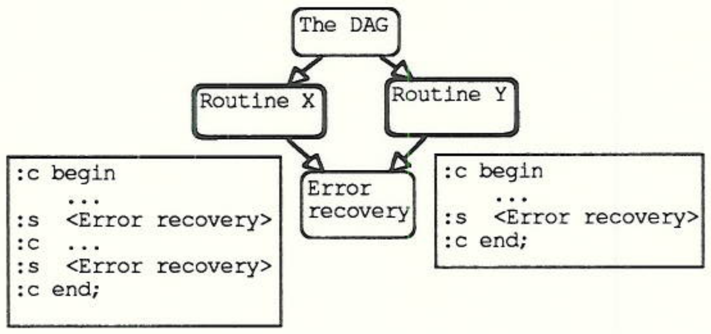
```
produce_output()
{
     int c, p;
     c = fgetc(fin);
     while (c != EOF) {
          if (isalpha(c)) {
               ungetc(c, fin);
               get_word(temp_word, fin);
               p = lookup(temp_word);
               if (p == 0)
                    fputs(temp_word, fout);
               else
                    fputs(symtable[p].style_ptr, fout);
               }
          else
               fputc(c, fout);
          c = fgetc(fin);
          }
}
```

C. GDDL - The Graphical Document Descriptive Language

The example document presented above was generated by applying the document generation routine of HSD to the GDDL specification of the program presented in Fig. 1. Each node in the GDDL is a pair of rectangles; the *name rectangle* stores the name of the node, the *text rectangle* stores the text of the node. Name rectangles always appear on the screen, text rectangles appear or disapear from the screen with a mouse action. The document is generated by a pre-order traversal of the GDDL specification, and can be decorated with LaTeX[10] commands for type-setting purposes.

The method employed to generate codes from GDDL is not as obvious. Let's examine GDDL more closely. The ':' symbols that appear on the left margin of the text rectangles in Fig. 1 have the following meanings: *:d* is followed by documents; *:c* starts a code region; *:b* defines a box to be appended with codes; *:a* adds codes to a box; *:s* is followed by the name of a subsection. The code generation routine traverses the tree in two passes. During the first pass, the *:b* constructs are used to define boxes. These boxes are subsequently appended with codes followed by the *:a* constructs. During the second pass, the code generation routine emits codes collected in the boxes and codes followed by the *:c* constructs. Instead of traversing the tree in pre-order, the code generation routine traverses the tree according to the direction of the *:s* constructs. For example, after encountering the *:s <The main program>* construct, the code generation routine starts processing *The main program* subtree. It returns to the position where the *:s <The symbol table>* can be read next after generating codes for the main program. A stack is used to keep track of the *return addresses* while traversing the tree recursively. With this traversing scheme, a macro, nestedly defined as a tree, can be called as many times as needed by its ancestor(s).

With more than one ancestor for some nodes, the structure of a GDDL is a direct acyclic graph. To handle the DAG properly, the document generation routine needs to be modified slightly: it needs to know whether a node has more than one ancestor or not. If a node has more than one ancestor, it only needs to write the text of the shared node once and generate a message such as "Please see section 1.1.1." for subsequent references. Fig. 2 shows an example of a hypothetical program represented as a DAG and its corresponding document.



```
:c begin
    ...
:s  <Error recovery>
:c  ...
:s  <Error recovery>
:c end;
```

```
:c begin
    ...
:s  <Error recovery>
:c end;
```

```
1 The DAG
     ...
1.1 Routine X
     ...
1.1.1 Error recovery
     ...
1.2 Routine Y
     ...
1.2.1 Error recovery
   Please see section 1.1.1.
```

Fig. 2

An HSD user can develop the GDDL specification by repeating the following steps:

1. Creates a node with a menu command and give it a name.
2. Connects the node to some ancestors or descendants.
3. Creates the corresponding text rectangle and fills it with text.

Since the system does not require any order in creating the nodes and the connectors, the specification can be created either top-down or bottom-up or a mixture of both. Although HSD is not designed for any particular design methodologies, I have tested the system to write programs designed with two structured design methods: JSD[8], and the top-down design and step-wise refinement[18]. My experience is that the HSD expresses the logical structure of the design without difficulties.

HSD is based on a user interface tools set developed for the Macintosh by the Meta Software Corporation[12]. Menu items are available to create nodes and connectors; graphical objects can be moved, resized, and deleted; standard Macintosh text editing facility is also supplied. The availability of the tools set substantially decreases the development time of HSD.

D. How HSD Relates to Studies of Programmers' Behavior

In [14], Soloway recommends the use of cognitively-based methodology for designing languages, environments, and methodologies. He suggests studing programmers' behavior before designing supporting tools for the programming process. For example, a better understanding of how programmers actually read and comprehend programs could provide valuable insights for designing tools to improve the readability of computer programs. The design of HSD is based on many results of empirical studies on programmers' behavior.

Weiser[16] proposes that the "slice" be used as the basis of programmers' mental representation when debugging an unfamiliar program. Conceptually, a slice is a collection of statements, often scattered throughout a program, that works together to achieve a goal. HSD allows programmers to construct *conceptual slices* to aid comprehension. Section 1.2.2.2 of the example program shows a conceptual slice.

A number of models and experiments[1,2,3,4,7,11,13,17] have been developed that explain how programmers comprehend programs. These models and experiments suggest that, at some stages of the comprehension process, programmers mentally divide a computer program into a number of smaller units called *chunks*. The program is comprehended by understanding individual chunks and the relation between chunks. I believe that writing a program as a collection of chunks is much more efficient than trying to reconstruct them while reading the program. HSD encourages programmers to compose a program as a collection of chunks.

Soloway[15] suggests that, in addition to syntax and semantics, students of introductory programming courses need to be taught the problem solving strategies that expert programmers use in programming. He proposes that explicit names be given to those strategies. The intent is to give instructors the means to talk about how problems are broken down into subproblems and about how strategies are combined to form *plans* to achieve the *goals* of a program. Soloway's insight is that it is important to disclose the *goal-plan* structure behind computer programs to students. I believe, in order to improve the readability of computer programs, professional programmers should be given means to disclose the goal-plan structure of their programs. The WEB system[9] as well as hypertext[6] techniques also provide ideas about what HSD should look like.

E. Possible Directions for Future Research

Although I am confident that using HSD to write programs' can improve their readability, more empirical studies are needed. The symbol table in the example would be much easier to understand if a picture illustrating its structure were included as part of the example document. A better program documentation system should clearly provide the user with tools to draw data structures and control flows. In [5], Brown demonstrates that algorithm animation can be used to illustrate the dynamic behavior of programs. It should be possible to include, in a program's document, pictures taken of various stages of a program's execution.

HSD should be viewed as a prototype whose purpose is to demonstrate the ideas presented in this paper. I hope this paper can generate more interest in developing complete systems which are integrated with compilers, run-time systems, and other tools for the entire programming process. The availability of such systems to professional programmers as well as to students is an important step toward literate programming.

The programming style presented in this paper is very different from conventional ones. Should the style become widely accepted, it would then be worthwhile to develop new programming languages which better fit the style.

The modularity of a computer program is an important indication about the programs quality. By representing a program's documentation as a tree might give us some opportunities to further analyze the program's modularity. For example, a chunk that uses variables declared by itself and its direct ancestors might be considered as having better modularity than a chunk that uses variables declared by its uncle or niece.

In [9], Knuth writes the following:

> ...I think that a complex piece of software is, indeed, best regarded as a *web* that has been delicately pieced together from simple materials. ...a programmer can now view a large program as a web, to be explored in a psychologically correct order ...

I agree that a program can be thought of as a web; however, I also believe that a program's document is best represented as a tree. Analyzing the cross-reference structure of the tree might give us some indications about how well the web was weaved and might allow us to give a more rigid measure about what the *psychologically correct order* means.

Acknowledgements

I thank Chris Haynes and Dennis Gannon for their encouragement. I also thank Steve Smith, Shinn-Der Lee, Bill Wang, and anonymous referees for their comments on this paper. Finally, I am grateful to David Drucker and Robert Shapiro for their help in using the *design* user interface.

References

[1] B. Adelson, "Problem Solving and the Development of Abstract Categories in Programming Languages," *Memory and Cognition,* Vol. 9, No. 4, 1981, pp. 422-433.

[2] M. E. Atwood and H. R. Ramsey, "Cognitive Structures in the Comprehension and Memory of Computer Programs: An Investigation of Computer Program Debugging," Englewood, Colorado: Science Applications, Inc., 1978, (NTIS No. AD A060 522).

[3] M. E. Atwood, A. A. Turner, H. R. Ramsey, and J. N. Hooper, "An Exploratory Study of the Cognitive Structures Underlying the Comprehension of Software Design Problems," (Technical Report 392), U.S. Army Research Institute for the Behavioral and Social Studies.

[4] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies,* Vol. 18, 1983, pp. 543-554.

[5] M. H. Brown, "Exploring Algorithms Using Balsa-II," *Computer,* Vol. 21, No. 5, May 1988, pp. 14-36.

[6] J. Conklin, "Hypertext: An Introduction and Survey," *Computer,* Vol. 20, No. 9, September, 1987, pp. 17-41.

[7] J. S. Davis, "Chunks: A Basis for Complexity Measurement," *Information Processing and Management,* Vol. 20, 1984, pp. 119-126.

[8] M. A. Jackson, *Principles of Program Design.* Academic Press, New York, 1975.

[9] D. E. Knuth, "Literate Programming," *The Computer Journal,* Vol. 27, No. 2, 1984, pp. 97-111.

[10] L. Lamport, LaTeX User's Guide & Reference Manual. Addison-Wesley, 1986.

[11] K. B. McKeithen, J. S. Reitman, H. H. Rueter, and S. C. Hirtle, "Knowledge Organization and Skill Differences in Computer Programmers," *Cognitive Psychology,* Vol. 13, 1981, pp. 307-325.

[12] Meta Software Corp., *design* Open Architecture Development System Reference Manual. Cambridge, Massachusetts, 1987.

[13] B. Shneiderman and R. Mayer, "Syntactic/Symantic Interactions in Programmer Behavior: A Model and Experimental Results," *International Journal of Computer and Information Sciences,* Vol. 8, No.3, 1979, pp. 219-238.

[14] E. Soloway, "A Cognitively-Based Methodology for Designing Languages/ Environments/ Methodologies," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* Pittsburgh, Pennsylvania, 1984, pp. 193-196.

[15] E. Soloway, "Learning to Program = Learning to Construct Mechanisms and Explanations," *Communications of the ACM,* Vol. 29, No. 9, September, 1986, pp. 851-858.

[16] M. Weiser, "Programmers Use Slices when Debugging," *Communications of the ACM,* Vol 25, No. 7, July, 1982, pp. 446-452.

[17] S. Wiedenbeck, "Beacons in Computer Program Comprehension," *International Journal of Man-Machine Studies,* Vol. 25, 1986, pp. 697-709.

[18] N. Wirth, "Program Development by Stepwise Refinement," *Communication of the ACM,* Vol. 14, No. 4, April 1971, pp. 221-227.