Requirements for an Elucidative Programming Environment

Kurt Nørmark
Department of Computer Science
Aalborg University
Denmark
normark@cs.auc.dk

Abstract

The main goal with this paper is to motivate and coin a variation of literate programming which we call elucidative programming. Elucidative programming is oriented towards program explanation with the purpose of throwing lights on important and complex program relationships. Since proposed by Knuth in 1984, literate programming has been one of the most viable approaches to a radical improvement of internal program documentation. Unfortunately, most programmers find the ideas of literate programming, as supported by WEB-like tools, for impractical, academic, and far-fetched in relation to current programming practices. With elucidative programming we intend to focus on the best ideas of literate programming. We disregards the aspects of publishing programs as technical literature, and we provide for mechanisms with which documentation can be added to a program without affecting or disturbing the source program. Our ideas about elucidative programming are presented as a number of requirements, and in a discussion of programming environment issues in relation to the new ideas.

1 Introduction

There are many different kinds of program documentation: *Pre-programming documentation* in terms of analysis and design documents, *user documentation* such as the user's manual of a computer application, *interface documentation* of public procedures in libraries, and *internal program documentation* which explains the interior of a program. Other classifications of documentation exists, such as the classification by Sametinger [24].

Work on program understanding can be categorized in at least two different groups: The *prevenient* and the *posterior* approaches. Using the prevenient approach we document the program understanding before, or side by side with the development of the program. Posterior approaches deal with extraction of program understanding from existing programs.

In this work we are interested in a prevenient approach to internal program documentation. More specifically, we study internal program documentation that represents an *understanding* of the program, as it is present among the programmers who implement the program. It is often the case that the program understanding is present at the moment the program is written, but disappears without being formulated in any written documentation. This is a major problem if or when we want to modify the program, because most modifications require the original understanding to be recovered. In section 2 of this paper we will discuss a number of reasons why programmers refrain from documenting the program understanding.

In section 3 we go on with a discussion of *literate programming*. We consider literate programming as one of most viable solutions to the challenge of internal program documentation. We argue that literate programming, and in particular the most dominating tool support of literate programming, suffers from a number of inherent weaknesses. As a consequence of these we propose to distinguish between literate programming on the one hand, and a more pragmatic variation of literate programming which considers the need of the everyday programmer who wants to maintain his or her program understanding. We will use the term *elucidative programming* for this variation. According the dictionaries the verb 'elucidate' means 'to make clear or plain, especially by explanation', and 'to throw light on something complex'. As such we find that this term hits the flavor of 'explanation' that we go for in our work.

In the main part of the paper (section 4 and 5) we discuss requirements for elucidative programming, and we describe an implementation of a programming environment that supports elucidative programming in Scheme [11].

2 Program Understanding

Programming is a creative and demanding activity. Programming requires a deep and systematic understanding of the underlying problems and their solutions.

Some of this understanding is represented in pre-programming documentation (such as requirement, analysis, and design documents). However, of practical reasons, very little of this documentation is ever updated in later phases of the program development process, or even less during the years of maintenance and further development. As a consequence, the pre-programming documentation tends to loose its value during the life time of a program. The growing interest in 'reverse engineering techniques' [29] (where the original design information is extracted from the program source) can be seen as an evidence of this.

A substantial amount of understanding turns up during the implementation of a program. At implementation time we are in touch with the details which often matter more than expected, and which tie the more overall program elements together. The understanding is embedded into program details, but only rarely it is formulated as documentation in clear and natural language. We see this as a major problem. In principle it would be a minor effort to write down and represent the program understanding. In real life, however, there are a lot of excuses for not doing so, and we will examine these below. If the understanding of the program is not written down in plain and natural language we have to extract it from the program source text when it is needed later on in the program's life time. However, it is most often a difficult, time consuming, and painful task to do so.

It is interesting to analyze why the program understanding is not written down while the understanding is present among the programmers. We can see the following reasons:

- **The program comment problem.**
  If the understanding is represented as program comments the program often seems to disappear between explanations. Program comments are not good for long and voluminous explanations. Programmers want relative clean source programs.

- **The program-documentation relation problem.**
  If the understanding is written in separate documents or files it requires the definition of strong relations among program fragments and pieces of documentation. Without such relations program and documentation will never be kept up-to-date. It is a nontrivial task to maintain such relations, and the maintenance process demands support from the program development tools.

- **The mental loading problem.**
  Programming is a mentally demanding activity which most often loads the programmer one hundred percents. If the documentation tools require unnecessary "mental overheads" the programmer will resist using them.

- **The programmer's motivation problem.**
  The programmer is not motivated to document his or her program understanding because the documentation efforts bring very little *immediate payback*. Most documentation efforts are long term investments in relation to program maintenance.

In order to overcome these obstacles we need a programming environment which alleviates some of the problems. In this paper we will outline the requirements for such an environment. But first we will discuss the observations from above in relation to the documentation style known as literate programming.

3 Literate Programming

Literate programming, as coined by Knuth [13] in 1984 is probably the best and most radical approach to internal program documentation. The main idea behind literate programming is to consider a program and its documentation as literature which are read by programmers in the same way as technical papers. Thus, with literate programming an analogy is drawn between published technical literature and programs.

Knuth implemented the WEB system [12, 15] as a set of tools which supports literate programming. The tools in the WEB system are called *weave* and *tangle*. Weave produces printable documentation, and tangle extracts and assembles the program fragments from the documentation. WEB was originally oriented towards Pascal programming and documentation written in TeX. After WEB a number of similar systems appeared [1, 30, 10]. The main variations stem from the programming and documentation languages supported. Some systems are independent of either the programming language or the documentation language.

In the literature a number of small examples of literate programs are available [5, 8, 9, 7, 32, 28]. There are also a number of papers which discuss different approaches to literate programming [2, 22, 23, 4, 25, 21, 6] and there exists a published annotated bibliography of literate programming [26].[1] In this paper we will not go further into these. Instead we will characterize WEB-like literate programming tools in relation to the discussion of program understanding from above.

[1] The most complete and up-to-date bibliography is available via Nelson H. F. Beebe's home page from `http://www.math.utah.edu/pub/tex/bib/litprog.html`.

First, with respect to the program comment problem, literate WEB programming contributes with a noteworthy approach. In a literate program, pieces of programs are annotations of the explanations, whereas in "ordinary programs" the explanations (in terms of comments) are annotations of the program. This change of organization, from program structure to a focus on the documentation structure, is one of the main contributions of the literate WEB programming tools. With this, it is possible to structure the "program source" with respect to its explanation rather than to the organization prescribed by the rules of the programming language. But, as we will see below, the price paid for this quality is relatively high.

Second, the WEB solution to the program-documentation relation problem is rather special. As seen above, the solution is to embed program pieces into the documentation. Thus, a program piece P is contained in the piece of text T which serves as the explanation of P. This provides for *proximity* between the program and its explanation which is both important and characteristic for the WEB approach to literate programming. The textual containment of the program in the documentation requires a linguistic framework which ties the documentation and the program together (and further on, a mechanism which allows us to assemble the program pieces to a whole program). In the following we will talk about an *interconnection language* for this purpose.

Third, the mental load of using a WEB system is high. This is partly due to the use of three languages in one document: The programming language, the documentation language, and the interconnection language. It takes a real expert (or long training) to master all three languages simultaneously while keeping the main mental efforts concentrated on problem solving. And furthermore, the fact that the source as seen by the compiler is different from the source as seen by the programmer, will often cause problems when the programmer is locating syntax and run time errors.

Fourth and finally, the beautiful documentation produced by the weave tool is intended to be the programmer's motivation for using a WEB-like literate programming tool. Almost everyone, who have seen the weaved output of a literate programming tool, likes it. But there are a number of problems with the approach. Most severe, the format available to the programmer during the programming and the understanding process is as ugly as the weaved output is beautiful. In other words, during the problem solving phase the programmer has to fight against rather old-fashioned, notational problems. The immediate gains to the programmer are almost zero. Furthermore, the weaved output is almost exclusively oriented towards a paper representation. Using today's media, a more on-line oriented representation using hypertext concepts would be a big gain.

In summary, we find that the ideas behind literate programming are very important, and still among the best in trying to find solutions to the problem of internal program documentation. However, literate programming, as envisioned by Knuth, is primarily oriented towards publication of programs as technical literature (such as [14]). This is different from the needs of the practical software engineer, who has to document his or her program contributions for fellow and future team members. As a consequence, we recommend that the area is split into two branches: *literate programming* for the publishing of programs as technical literature, and *elucidative programming* for documenting the understanding of practical programs in a software development project.

4 Elucidative Programming

We will now describe the ideas of elucidative programming in more details. We do that by enumerating a number of requirements for an elucidative program. During the discussion of these requirements we will make contrasting comparisons to the ideas behind literate programming.

Requirement 1: *The internal documentation must be oriented towards current and future developers of the program.*

We find it important to stress, as the first requirement, who are going to benefit from an improved internal documentation. The target group of the internal documentation in an elucidative program is the current and future developers of the program. Thus, the documentation is not meant to be 'literature' for the general public, nor to be educational material in some sense. In this respect, elucidative programming is quite different from literate programming, in which the publication aspect is emphasized. With elucidative programming we want to encourage the program developers to write down sufficient information for fellow programmers—current and future—to understand the program. In that way further development of the program is made possible without costly "detective work" on the source code using a posterior approach (cf. section 1).

Requirement 2: *The internal documentation must address explanations that maintain the program understanding and clarify the thoughts behind the program.*

It is natural, as the second requirement, to state requirements for the nature and the purpose of the documentation in an elucidative program. An elucidative program is intended to represent the source program as well as explanations, which retain the understanding of the program, as present among the programmers who implement and maintain it. This includes relevant background information and rationales, which may be important to be aware of if and when the program has to be further developed.

As a contrast to the first requirement, this requirement

does not separate elucidative and literate programming further from each other. On the contrary, we find that the two approaches coincide on this requirement.

It is appropriate to point out an important, derived benefit of systematic formulation and representation of the program understanding. Knuth argues that we can expect a quality improvement of the program if the programmers expose their understanding of the program explicitly [13, Economic issues]. The improvement manifests itself in fewer errors and less time used to debug the program. This observation plays an important role in the overall discussion about resources and time used for programming versus internal program documentation.

Requirement 3: *The program source file must be intact, without embedded or surrounding documentation.*

For most programmers, and in relation to many, existing program processing tools (not least compilers) it is highly desirable to retain the source files and their organization intact when we practice elucidative programming. This is in contrast to literate programming, where the aggregated WEB description plays the role of a "new source program". Using the WEB approach, a separate tool is necessary to extract and assemble the program in order to prepare for compilation. This gives rise to a multitude of problems. First and foremost, any reporting of errors which relates to the source program (as seen by the compiler or interpreter) needs to be mapped back to the "new source". Second, the tangle tool must produce a certain structure of source files, organized in particulars directories and files (such as Java files and packages). Third, when programs become large, there will be a need to split the "new source" into modules, hereby leading to a new layer of organization. Fourth and finally, some tools uses time stamps for automatic production of derived files. This becomes difficult to deal with in case all "old source files" are made automatically from a "new source file".

As a consequence of these observations (all of which are well known from the literate programming literature cited in section 3) we will require that the ordinary and common concept of the *program source file* is retained in elucidative programming. The program aspects and the documentation aspects are separated, typically into different files. We therefore need to introduce relations which keep pieces of program and their documentation in close correspondence. From the discussion above, we have rejected the use of a containment relation for this purpose. There are many different ways to realize the relations between program and documentation, but at this level of the discussion it would be inappropriate to impose further requirements.

Requirement 4: *The programmer must experience support of the program explanation task in the program editing tool.*

An elucidative programming environment cannot be realized through a single, transformation tool, like tangle or weave in WEB-like literate programming. It is necessary to support the elucidative programming task at several levels, starting with the program editor. It will be a requirement that the program editor can help the programmer to handle the separation of program and documentation, and the necessary relations among the two of them. Thus, it should be easy to navigate between program and documentation using the editor. Also, the creation of relations between the documentation and a program should enjoy specialized editor support. In that way we intend to minimize the mental load imposed on the programmer in such a way that the programmer can concentrate on his or her main task: To create a well-documented program which solves a given problem.

Besides the editor support, there will in most programming environments be a need for a tool which can process the program and documentation in order to prepare the internal program documentation. (This is addressed in further details in the sixth requirement). This tool is the functional equivalent of the weave tool in WEB-like literate programming environments.

Elucidative programming will be most powerful and satisfactory if the tools, which support it, know some details of the programming language. As an example, which is relevant at editor level, it is an advantage if the editor is aware of the explained program units. In that way it is possible to provide for a smooth interaction during the process of relating program pieces with their documented understanding. It is also relevant to identify applied names together with their relations to defining name occurrences in the program. These aspects of the program editing tools call for parsing and static analysis of involved programs.

Requirement 5: *The program "chunking structure" follows the main abstractions supported by the programming language.*

In the literature about literate programming a *program chunk* (or scrap) denotes a piece of program which we chose to describe or explain. Such chunks can be selected arbitrarily, from very fined grained structures to the entire program. Usually, a program chunk is a syntactically well-defined program fragment. In WEB-based literate programming the named chunks are used to assemble the program from a large number of pieces.

The vast majority of chunks in modern programs follow the abstractions as provided for by the underlying programming language. In other words, most program chunks correspond to procedures, functions, classes, modules, and similar units. It is clearly an advantage to represent the chunks as named abstractions. Hereby we are freed from dealing with double naming in terms of chunk names and names of the abstractions.

Literate programming is an exceptional vehicle for documentation of a structured, stepwise refinement development

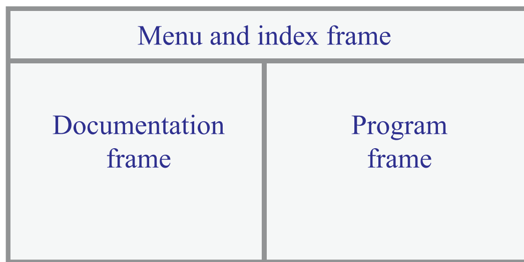

Figure 1. The browser layout of an elucidator.

process [31]. We will, however, like to detach elucidative programming from this heritage, by requirering that the units of documentation correspond to the major abstractions found in a program. This makes the documentation efforts much more manageable (especially in an environment with language knowledge) and we do probably not loose many chunks (candidates of good documentation) in well-structured, modern programs.

Requirement 6: *The documented program must be available in an attractive on-line represention suitable for exposition in an Internet browser.*
Even though the programmer will experience a considerable support of elucidative programming while using the program editor, it will for most environments be a requirement that the program and documentation can be exposed in an attractive format on intranets or the Internet. We envision an exposition somewhat similar to JavaDoc's interface documentation of classes in the Java language.

In comparison with literate programming, our emphasis has changed from beautiful programs exposed in a book to attractive presentations in an Internet browser. We feel this is a natural development over the last 15 years, since the introduction of literate programming in 1984.

5 Programming Environment Support

Given the requirements from section 4 we will now discuss how elucidative programming can be supported via tools in a programming environment. We start with a discussion of the basic model behind the tools in the environment. At the concrete level we will describe a particular implementation of the tools which supports elucidative programming in the programming language Scheme. More details about the Elucidative Scheme environment can be found in a separate paper [20].

5.1 Concepts and Model

In order to keep the parts of an elucidative program together we introduce the concept of a documentation bundle. A *documentation bundle* consists of a number of programs, a documentation unit, and a setup description. At the concrete level these are all files. The setup description defines the parts of a documentation bundle (in order to keep the parts together) and it allows for setting various processing options of the elucidator tool (see section 5.2).

We use a simple documentation model. All the documentation is represented in a single text file. Besides a header part with information about title, author, etc., the structure of the documentation consists of sections and subsections (entries). Each entry is supposed to explain and document a single aspect of the program, and it may involve one or more program units. In this respect, the overall structuring of the documentation is similar to Knuth's original WEB system.

From each documentation entry it is possible to make *relations* to a number of definitions in the documentation bundle's program files. We distinguish between strong and weak relations. *Strong relations* are used for explanation of program details. Thus, if we have an instance of a strong relation between a documentation entry E and a program definition D, details in D are explained in E. *Weak relations* are used where program definitions are mentioned without being explained.

According to requirement number five from section 4 the explained program units correspond to the main abstractions in the programming language. As mentioned earlier, these are typically modules, classes, methods, functions, or procedures. The relations are transformed to hypertext links by the elucidator tool which produces the on-line presentation of the documentation bundle (see section 5.2). Consequently, it is easy to navigate from the documentation to the program unit being explained or mentioned.

As discussed in section 4 we do not include program fragments in the documentation, neither do we present fragments of the program inside the documentation. This is one of the main differences between an elucidative programming tool and the literate programming WEB tools. One of the main ideas behind the elucidative programming tools is to present documentation and program beside each other (see figure 1) with flexible means of *mutual navigation* in between them.

The relation between the sections of the documentation unit and program units is also used to generate links in the other direction, namely from program units to the sections in the documentation in which they are explained.

Both the documentation and the programs are structured hierarchically by means of textual embedding. Besides these basic hierarchies there are cross reference relations internally in the documentation, and internally in the programs. At the documentation side it is possible, in a simple and straightforward way, to relate one section to another. This gives rise to cross referenced documentation entries

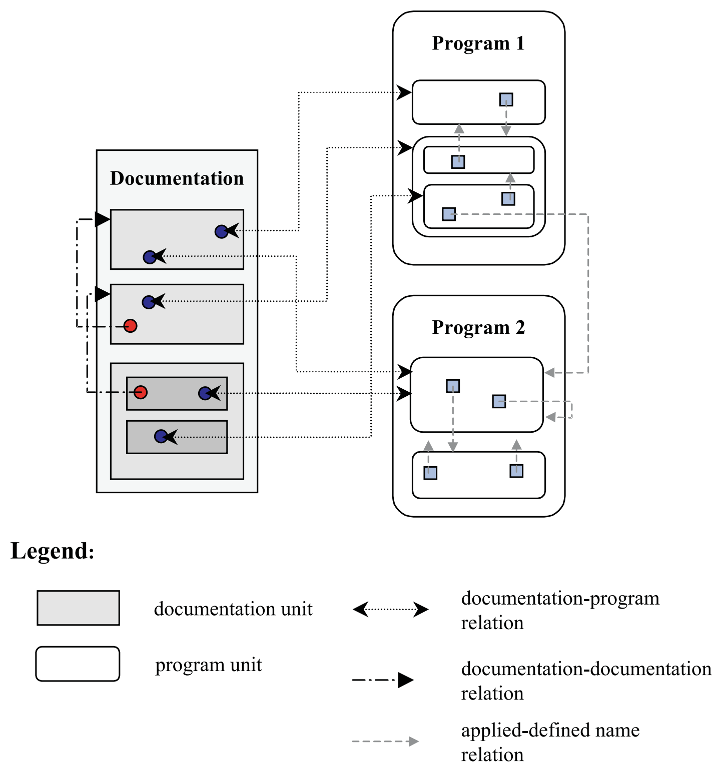

Figure 2. An illustration of the elucidative programming model.

in the browser presentation. At the program side each applied name is related the corresponding definition, provided that the relevant definition is contained in the documentation bundle. Establishing this relation requires knowledge of the programming language syntax. Thus the elucidator tools will be language dependent in the same way as most literate programming WEB tools depend on a particular programming language.

Figure 2 summarizes this model in terms a drawing which illustrates a documentation bundle containing a documentation unit, two program units, and a number of relations in between them.

Using the relations introduced above it is possible to refer to individual program units (definitions) from the documentation, but it is not possible to refer to particular parts of, or places in, the program units. When we document larger units, or if we need to refer to specific details in a unit, our approach falls short. Consequently, we have introduced a *source marker* concept, via which we can identify an exact position in a named program unit. Source markers can be used in the documentation to direct the user's attention to the location of the source marker. A source marker in the documentation is associated with the closest preceding anchor point of a strong relation. Source markers in the program is, quite naturally, associated with the containing definition. Source markers are shown as colored dots in both the documentation and programs. Source markers can be navigated in both directions, in the same way as the other relations. Figure 3 shows an example of documentation and a program with source markers.

The introduction of source markers compromises the third requirement, which states that the source program must be left intact. Lexically, the source markers have to be placed in program comments in order not to interfere with the programming language syntax. Concretely, a source marker is represented as @x, where x is a single character, and as such the source markers do not "disturb" the program source in any significant way. (See figure 4 for an example of a program text that uses source markers). According to our experience with the Scheme Elucidator, the source markers provide for a very close coupling between the documentation and the documented programs.

In imperative Scheme programs there may be top-level constructs or program sections which are not definitions. There is a natural need to explain such constructs or sections in the documentation. However, using the concepts from above, there is no way to refer to such forms. We have therefore introduced the concept of sectional comments. A *sectional comment* introduces a name for a particular construct or section in the program. Via use of source markers it

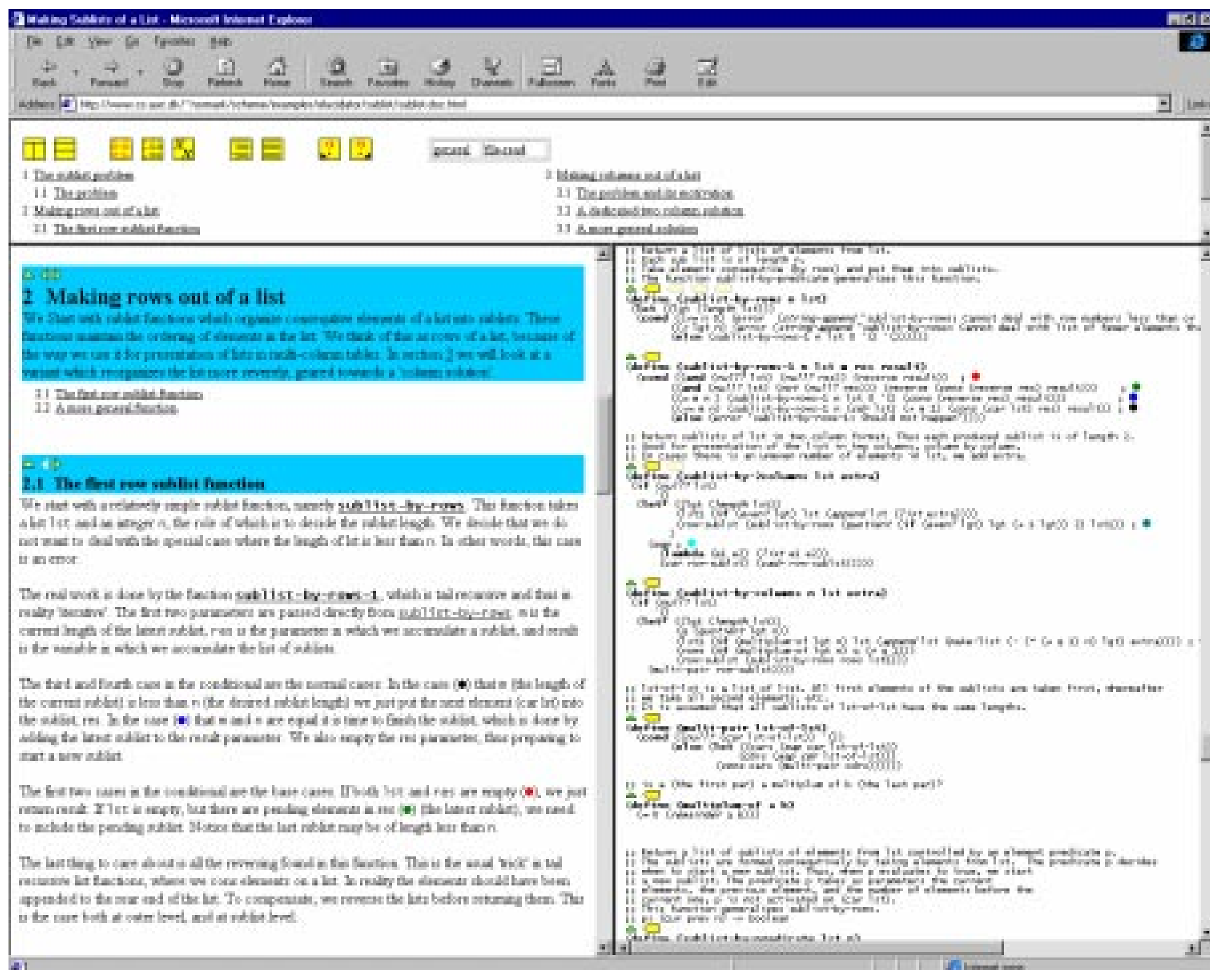

Figure 3. A snapshot of an elucidative Scheme program presented in an Internet browser.

is possible to direct the attention to details within a section.

5.2 The Elucidator tool for Scheme

The main tool in an elucidative programming environment, called the *elucidator*, creates an on-line presentation of a documentation bundle which can be shown in a WWW browser. As illustrated in figure 1 we use a three frame layout, where the top frame contains the main navigation buttons and room for various indexes (shown on demand), the left bottom frame presents the documentation, and the right bottom frame shows one of the program files in the documentation bundle. Each of the frames can be enlarged or decreased relative to the other two frames.

The elucidator supports *mutual navigation* between the documentation frame and the program frame. If the user clicks on a program reference in the documentation frame, the definition of the program unit will be shown in the program frame. And similarly, if a user clicks on a documented program unit in the program frame, the section in which the unit is explained will be shown in the documentation frame. (Documented program units are identified by one or more "left arrows" which indicates the existence of program explanations in the documentation frame). In addition, all applied name occurrences in the program frame can be used for navigation to the definition of the name, provided that the definition is contained in the current documentation bundle. Figure 3 shows an example snapshot of an elucidative Scheme program.

The upper menu and index frame can be used to produce a number of useful indexes:

- A full or compact index of sections and subsections in the documentation (table of contents).
- A definition index which enumerates all the defining name occurrences in the documentation bundle.

Figure 4. A snapshot of the Emacs editor during the work on the example shown in figure 3.

- A cross reference index which enumerates the definitions in which all applied names occur.

The definition index and the cross reference index may be broken into alphabetic pieces in order to provide for quick loading and flexible navigation (without excessive scrolling). The indexes produced by the elucidator are similar to the indexes made by most of the literate programming WEB tools. Besides presenting the indexes, the upper frame can be used to control which program source file is shown in the program frame.

5.3 The editor part of the environment

Requirement number 4 from section 4 calls for editor support of an elucidative programming effort. We use a plain text editor (Emacs) for the Scheme elucidator. The main idea is to work on the documentation and a program using a split screen layout, such that both the documentation and one of the programs are visible in an editor window. Figure 4 shows the editor in a situation which corresponds to the elucidator snapshot in figure 3.

As illustrated in the figure, the structural units in the documentation are described using a very simple, special purpose markup. If additional formatting is needed within the sections it is possible to use HTML tags. This “mixed approach” is similar to the solution used for JavaDoc. From the figure we also see that the relations between the documentation and the program are defined by mentioning names of Scheme functions in curly brackets. If necessary, we can use a two level naming scheme which involves a source file name and a name of a Scheme definition in the file. An optional modifier character (such as ‘*’, ‘+’, or ‘-’) after the opening bracket determines the kind of relation (strong, weak, or just typographic emphasis).

As the first issue, it is natural and very helpful if the editor knows the constituents of a documentation bundle, including their roles (whether setup, documentation, or pro-

gram). We support a *setup command* which brings all parts of a documentation bundle up in the editor. The setup command also marks the elucidator relevant buffers in order to provide for special functionality on these.

Navigation between documentation and programs are supported in the editor as well. More specifically, a generic navigator command is able to find a documentation entry or a program unit in "the other editor window". In that way it is easy and flexible to find a documented program unit from the documentation text, or the relevant explanation from a program unit. From a navigation point of view the editor part of the elucidator is quite similar to the elucidator presented in a browser. The main difference is the visual clues and the decorations used.

The definition of relations between the documentation and a program unit is also supported in the editor. The editor has knowledge of all definitions in the documentation bundle (including the source file in which they reside). A textual completion mechanism makes it easy and secure to define a new contribution to the relation between the documentation and the source program.

Finally, the editor supports a variety of textual templates for documentation sections such that the programmer does not need to remember syntactical details of the documentation format. A particular editor command is able to create the needed directory structure, which contains an HTML directory, a directory with icons, an internal bookkeeping directory, etc. In that way it easy to initiate a new documentation bundle.

Internally, the editor part of the environment builds on the knowledge extracted from the last processing by the elucidator tool. Specifically, the editor reads some data structures which are generated by the elucidator. These data structures contain all the defined program names (including source file information), a list of all documentation sections, and a list of all source files. Using this approach, the editor's knowledge of the documentation bundle is not always fully up-to-date. However, we think that this approach is a reasonable and practical alternative to incremental parsing of the programs and documentation at edit time. A programmer using the editor part of the elucidator environment is motivated to frequent runs of the Elucidator tool in order to have reasonably up-to-date information about the documentation bundle while constructing or maintaining the program.

5.4 About the tool implementation

The Scheme elucidator tool has been implemented in Scheme itself using the LAML libraries (see below). The editor part of the elucidator is implemented in Emacs Lisp, which is the extension language of the Emacs editor. The elucidator processing can be activated via a single keystroke from Emacs, or via a Unix command.

The elucidator tool generates a set of HTML pages which represent the files of the documentation bundle in a WWW browser. This generation is greatly enhanced by the LAML libraries. LAML [18, 19, 17], which means *Lisp Abstracted Markup Language*, provides for generation of HTML pages via descriptions in Lisp (Scheme). LAML mirrors all HTML elements into Scheme. In that way, any HTML page can be generated from Scheme, using functional programming techniques. Furthermore we can make abstractions of HTML expressions and define customized mark up tags within Scheme, much along the lines of XML [3].

6 Status and conclusions

In summary this paper has coined a variant of literate programming which we call elucidative programming. We have formulated a number of requirements for elucidative programming which make up a contrast to literate programming. We feel strongly that the program source files and the documentation need to be separated. It is valuable to keep the source program intact and free of lengthy documentation. This is the main difference between an elucidative program and a literate program in the WEB tradition.

In WEB systems for literate programming the proximity between the documentation and the program is very strong. The reason is that program fragments are organized physically in its documentation. In our elucidative programming environment the proximity is weaker because it relies on relations (links) between units in two separate documents. Our organization makes it easier to document transverse aspects of a program which involve several different program units. Notice, however, that this organization may cause certain maintenance problems.

We have designed and implemented two elucidative programming tools for Scheme: An elucidator which produces WWW pages of a documentation bundle, and an editor tool (in terms of Emacs extensions) which supports the creation and modification of the documentation bundle. We are currently using the Scheme Elucidator to get experience with the idea of elucidative programming on LAML related software, which is written in Scheme. As an example of particular interest, we have used the Scheme Elucidator to document the current implementation of the tool itself.

A group of five master thesis students have implemented an elucidator for Java [27]. The Java elucidator follows the same principles as the Scheme elucidator. However, Java is a more difficult and challenging language to support than Scheme. This implies, for instance, that a more elaborate naming scheme is needed to address program entities from the documentation. From an implementation point of view the Java elucidator is also more advanced than the Scheme

elucidator described in this paper. The Java elucidator stores the result of a 'program abstracting process' in a relational database. Furthermore, the Java elucidator generates the HTML pages on demand (the Scheme elucidator generates a set of static HTML pages). More information about the Java elucidator can be found on the *elucidative programming home page* [16].

We have not reported on any substantial experience with elucidative programming in this paper. This will be done in forthcoming papers from our group. With this paper we have introduced the ideas, and we have reported on our current elucidative programming tools.

The software tools for elucidative programming, examples of elucidative programs and other results are available from the Elucidative Programming home page. The Scheme Elucidator is available as free software from the LAML home page on the Internet [17].

References

[1] P. Briggs. Nuweb, A simple literate programming tool. Technical report, Rice University, Houston, TX, USA, 1993.

[2] M. E. Brown and B. Childs. An interactive environment for literate programming. *Structured Programming*, 11(1):11–25, 1990.

[3] W. W. W. Consortium. Extensible markup language (xml) 1.0, February 1998. http://www.w3.org/TR/REC-xml.

[4] D. Cordes and M. Brown. The literate-programming paradigm. *IEEE Computer*, 24(6):52–61, June 1991.

[5] P. J. Denning. Announcing literate programming. *Communications of the ACM*, 30(7):593, July 1987.

[6] J. Hamer. Literate programming: a software engineering perspective. In *Software Education Conference (SRIG-ET '94): Proceedings, November 22–25, 1994, University of Otago, New Zealand*, pages 282–288, 1995.

[7] E. Hamilton. Literate programming: Expanding generalized regular expressions. *Communications of the ACM*, 31(12):1376–1385, December 1988.

[8] D. R. Hanson. Literate programming: Printing common words. *Communications of the ACM*, 30(7):594–599, July 1987.

[9] M. A. Jackson. Literate programming: Processing transactions. *Communications of the ACM*, 30(12):1000–1010, Dec. 1987.

[10] A. L. Johnson and B. C. Johnson. Literate programming using `noweb`. *Linux Journal*, 42:64–69, October 1997.

[11] R. Kelsey, W. Clinger, and J. R. (editors). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[12] D. E. Knuth. The WEB system of structured documentation. Technical Report STAN-CS-83-980, Department of Computer Science, Stanford University, September 1983.

[13] D. E. Knuth. Literate programming. *The Computer Journal*, May 1984.

[14] D. E. Knuth. *Tex: The Program*. Computers and Typesetting. Addison Wesley, 1986.

[15] D. E. Knuth and S. Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison Wesley, 1993.

[16] K. Nørmark. The elucidative programming home page. http://www.cs.auc.dk/~normark/elucidative-programming/, 1999.

[17] K. Nørmark. The LAML home page. http://www.cs.auc.-dk/~normark/laml/, 1999.

[18] K. Nørmark. Programming World Wide Web Pages in Scheme. *Sigplan Notices*, 34(12):37–46, December 1999. Also available via [17].

[19] K. Nørmark. Using Lisp as a markup language—the LAML approach. In *European Lisp User Group Meeting*. Franz Inc., 1999. Available via [17].

[20] K. Nørmark. An elucidative programming environment for Scheme. In *Proceedings of NWPER'2000 - Nordic Workshop on Programming Environment Research*, May 2000. Available via [16].

[21] K. Østerbye. Literate Smalltalk programming using hypertext. *IEEE Transactions on Software Engineering*, 21(2):138–145, February 1995.

[22] N. Ramsey. Weaving a language-independent WEB. *Communications of the ACM*, 32(9):1051–1055, September 1989.

[23] T. Reenskaug and A. L. Skaar. An environment for literate Smalltalk programming. *Sigplan Notices*, 24(10):337–345, October 1989.

[24] J. Sametinger. Object-oriented documentation. *Journal of Computer Documentation*, 18(1):3–14, january 1994.

[25] J. Sametinger and G. Pomberger. A hypertext system for literate C++ programming. *Journal of Object Oriented Programming*, 4(8):24–29, 1992.

[26] L. M. C. Smith and M. H. Samadzadeh. An annotated bibliography of literate programming. *Sigplan Notices*, 26(1):14–20, January 1991.

[27] S. Staun-Pedersen, M. R. Andersen, V. Kumar, K. L. Sørensen, and C. N. Christensen. The elucidator - for Java. Preliminary master thesis report, Janunary 2000. Available from http://dopu.cs.auc.dk.

[28] C. J. Van Wyk. Literate programming: An assessment. *Communications of the ACM*, 33(3):361, 365, March 1990.

[29] R. C. Waters and E. Chikofsky. Reverse engineering: Progress along many dimensions. *Communications of the ACM*, 37(5):22–25, May 1994.

[30] R. Williams. FunnelWeb user's manual. Technical report, University of Adelaide, Adelaide, South Australia, Australia, 1992.

[31] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[32] C. J. V. Wyk and D. C. Lindsay. Literate programming: A file difference program. *Communications of the ACM*, 32(6):740–755, June 1989.