An Elucidative Programming Environment for Scheme

Kurt Nørmark

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

Abstract. In this paper we describe a programming environment for Scheme that supports elucidative programming. Scheme is a programming language in the Lisp family. Elucidative programming is a variant of literate programming. Literate programming represents the idea of structuring a program as fragments that are contained in an essay that documents the program understanding. Elucidative programming is in a similar way based on the ideas of documented program understanding, but in contrast to literate programming, elucidative programming leaves the program intact. The relations between the documentation and the units of the program are defined without use of containment. In addition, the elucidative tools are oriented towards program presentation in a WWW browser.

1 Introduction

Program development is based on understanding. The understanding is embodied and encoded in the program. Unfortunately, it is not easy to recover the understanding from the program. Consequently, a great amount of efforts are used to reestablish the original understanding when the program needs updatings of various kinds. The widespread interest is reverse engineering tools, which flourish in program comprehension circles [30, 1] is a clear evidence of this observation.

In this paper we recommend an investment in documented program understanding. The essential understanding, present among the people who write the program, should be captured and related to the relevant program units. Seen in perspective of the program life time it is simply not economical to forget the program understanding, and to recover it repeatedly via detective work, which is very difficult to support by effective tools.

The ideas about documented program understanding are not new. Literate programming has been around for 15 years, without causing significant impact on everyday software development practice. Part of the reason is that literate programming[1] is extreme in several directions:

[1] When we in this paper discuss the practical elaboration of literate programming we have the WEB-like tools in mind [12, 15, 2, 31, 10]. However, the literate pro-

- It is based on the ideas of breaking the program into fragments that are contained physically in the document which represents the program understanding. Thus, the program "lives in" the documentation. The concept of a source file (which is familiar to most programmers) does not exists in the literate programming paradigm.
- It mixes fragments of a text formatting language and fragments of a programming language in one "ugly" and monolithic file, the value of which is low in the development situation.
- It aims at documentation with literate value which can serve as technical literature in the same way as scholarly papers.

It is our hypothesis that literate programming is beyond reach of the average programmer. The ambition of literate programming is too high, the program artifacts are too far from mainstream, and current programming environments will suffer too much if adapted to the 'literate ideas' in a WEB-like elaboration.

With elucidative programming we keep the basic idea of documented program understanding from literate programming. However, we re-orient the approach in the following ways:

- The source program is left intact, without embedded or surrounding documentation.
- The program understanding is described in a document which is firmly related to named units in the source program.
- The documentation is targeted at the team of current and future program developers. Hereby the documentation addresses, in a narrow way, the needs of the programmers in the team that are going to maintain the program. We are not interested in a program as a publication (article or book).

In this paper we will describe the tool impact of introducing elucidative programming in an environment that supports the programming language Scheme [11]. In that respect, we are concerned with two overall goals. First, we want to orient the tools toward the medium of the Internet, WWW, and HTML. We have witnessed the great success of documenting class library interfaces using this medium in the Java Development Kit [3]. We are eager to find out whether program documentation of more internal nature can be made by similar means. Second, we want to integrate the support of elucidative programming in an existing editing environment. With this goal programmers can continue "programming as usual", but now in a *documentation enabled environment.*

We are aware of several obstacles that need to be overcome in order for elucidative programming to succeed. The obstacles rely on positive answers to the following questions:

- Is it realistic to expect that programmers retain their program understanding in a free style story or essay?

gramming paradigm may be supported by many other kinds of tools. When applied appropriately, the tools in an elucidative programming environment may support literate programming as well.

1. The internal documentation must be oriented towards current and future developers of the program.
2. The internal documentation is intended to address explanation which serves to maintain the program understanding and to clarify the thoughts behind the program.
3. The program source file must be intact, without embedded or surrounding documentation.
4. The programmer must experience support of the program explanation task in the program editing tool.
5. The program "chunking structure" follows the main abstractions supported by the programming language.
6. The documented program must be available in an attractive, on-line representation suitable for exposition in an Internet browser.

Fig. 1. Requirements for an elucidative programming environment.

– Is it possible to keep the documented program understanding up-to-date and valuable during the life time of the program (in the maintenance phase of the program).

We are not presenting substantial answers to these questions in the present paper. However, these questions are central in our ongoing research, and future papers from our group are expected to address these important issues.

In section 2 we will introduce the ideas and the concepts of elucidative programming. In section 3 we present a concrete example of an elucidative Scheme program. A discussion of the tools in the elucidative Scheme programming environment follows in section 4. The paper ends with a description of related work, status of the research, and conclusions.

2 Elucidative Programming

To "elucidate" is to throw light on something complex and to make clear or plain, especially by explanation.[2] We introduced the idea of elucidative programming in an earlier paper [20]. In that paper we discussed elucidative programming and literate programming in relation to each other, and we came up with six requirements for an elucidative programming environment. These requirements are summarized in figure 1.

[2] This is the meaning of the verb "elucidate" according to *The American Heritage Dictionary of the English Language*. As such we find that this term hits the flavor of "explanation" that we go for when an understanding of a complicated program has be to written down. The word "elucidate" is the most attractive among several candidates such as "explain", "expound", and "explicate".

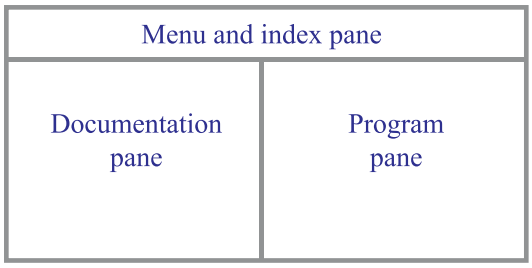

Fig. 2: *The layout of panes in an elucidator.*

2.1 User interface

From a user interface point of view, the central idea of the Elucidator is to present the program and the documented program understanding as hypertext in two relative large panes of a window, see figure 2. In that way the documentation may be presented in a pane on the left half part (or top) of the screen and the program in the right half part (or bottom) of the screen. A concrete example can be seen in figure 4, which will be discussed in section 3 of this paper. The proximity of documentation and program gained by this setup together with the mutual navigation in between the two panes make up the main characteristics of the user interface of an Elucidator.

2.2 Hypertext aspects

Seen as hypertext, the nodes of an elucidator are very coarse grained. The entire documentation is a single node in which sectional units are embedded into each other. Similarly, each source program file is represented in a node where the units of the programs are composed and embedded according to the rules of the programming language. In that respect, the elucidator runs counter to other hypertext-based programming environments [22, 21] where more fragmented models seem to dominate. The hypertext links are derived from relations among program and documentation entities. This is explained in more details in section 2.3.

2.3 Central model

From a modelling point of view, the program and the documentation are broken into *entities*. At the program side, the entities are the overall building blocks (named abstractions) of the program, such as classes, procedures, and functions. At the documentation sides, the entities are sections and subsections of the program explanation. As an integral part of the entity concept there must exist a *naming scheme* which allows us to refer to the program entities from the documentation, and vice versa.

In the Scheme environment, the most important program entities are top-level `define` forms, such as the function definition

```
(define (multiplum-of a b)
   (= 0 (remainder a b)))
```

If the `define` form is present in a file called `file`, the qualified name of this form is `file$multiplum-of`. In the current Scheme elucidator we do not support naming of more local definitions, although it would be straightforward (at least in principle) to generalize the naming scheme to deal with local definitions as well.

In addition, an arbitrary top-level form with a preceding *sectional comment* may be defined as an entity in a Scheme program. A sectional comment identifies one (or more) Scheme expressions. The following program fragment serves as an example:

```
; ::error-handling::
; Here we handle errors in the input.
(if (not (input-data-ok?))
(begin
  (write-page
     "Error messages''
     (string-append
       (font 4 red
        "There where errors in your input") (p)
       "Please try again!"
      ))
  (exit)))
```

A Scheme comment line starts with a semicolon, and a sectional comment name is enclosed in double colons. The Scheme form succeeding the sectional comment can be referred to by the qualified name `file$error-handling` if the error handling appears in `file`. It turns out that entities named via sectional comments are necessary for the proper documentation of Scheme programs with imperative constructs.

In the Scheme environment sections and subsections (called entries) in the documentation are identified and named with specialized markup. Here is an example of a section and an entry in that section:

```
.SECTION intro-section
.TITLE Introduction
.BODY
  Introductory text.
.END

.ENTRY attack-plan
.TITLE The plan of attack
.BODY
  Documentation describing the
  plan of attack.
.END
```

The names of sections and entries appear just after the `SECTION` and `ENTRY` keywords.

Program and documentation entities can be connected to each other by means of a few natural *relations*, all of which are binary. If we assume that P (together with P1 and P2) are program entities and D (together with D1 and D2) are documentation entities we can describe the meanings of the relations in the following way:

- **The *strong doc-prog* relation**:
  An element (D,P) in the *strong doc-prog* relation represents that the program entity P is explained in the documentation entity D.
- **The *weak doc-prog* relation**:
  An element (D,P) in the *weak doc-prog* relation represents that the program entity P is mentioned (without being explained) in the documentation entity D.
- **The *prog-prog* relation**:
  An element (P1,P2) in the *prog-prog* relation represents that P1 uses the entity P2.
- **The *doc-doc* relation**:
  An element (D1,D2) in the *doc-doc* relation represents that the documentation in D1 relies on the documentation in D2 seen from an elucidative point of view.

In addition, there is a relation which we could call *prog-lang*, which relates an instance of a Scheme language construct or standard procedure to its description in a hypertext version of the *Revised Report on the Algorithmic Language Scheme* [11].

In an elucidator tool each element in one of the relations mentioned above are represented by hypertext links. An element (D,P) in one of the *doc-prog* relation gives rise to two links:

- a link source anchored in a position of the documentation entity D, and destination anchored in the program entity P.
- a link source anchored in an icon just in front of the program entity P, and destination anchored in the documentation entity D.

An element in the *prog-prog* relation relates an applied name occurrence to its defining name occurrence. In order to be more precise, let us assume that (P1,P2) is an element in the *prog-prog* relation, that P2 is named N, and that (P1,P2) gives rise to a link L. L is source anchored at an applied occurrence of N in P1. The destination anchor of L is a presentation of P2, which defines N.

An element (D1,D2) in the *doc-doc* relation gives rise to a cross reference link from one place in the documentation to another section.

2.4 Source markers

The links derived from the relations described above can be used to connect sections in the documentation with named abstractions and sections in a program

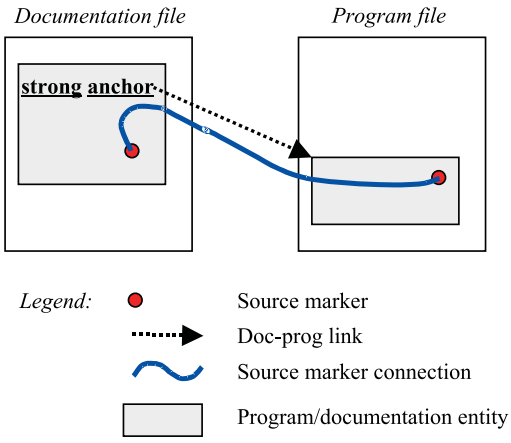

Fig. 3: *The connection of a source marker in the documentation through the anchor point of strong doc-prog relation instance to the corresponding source marker in the program.*

source file. However, in some explanations it is desirable to address finer details in the program. Of that reason we have introduced the concept of source markers. A *source marker* denotes a particular point in a program entity. Source markers must appear in comment positions in order not to interfere with the syntactic rules of the programming language. In order to minimize clutter in the program comments we use a minimal two character notation `'@<character>'` for source markers in a Scheme program.

At the documentation side, source markers may be used when we explain the program details next to a source marker in the program. A source marker in the documentation is associated with the anchor of the link corresponding to closest preceding strong `doc-prog` relation element. A source maker in the documentation is the source anchor of a link which goes to the corresponding sourcer marker in the program, and vice versa.

Figure 3 illustrates the connection of source markers in the documentation and the program via an element in a strong `doc-prog` relation.

2.5 Organizational aspects

The concept of a *documentation bundle* is central seen from an organizational point of view. The documentation bundle is described in a setup file which enumerates the program files in the bundle. The documentation is either inlined in the setup file as LAML expressions [18, 19] or more typically imported from a text file which uses the specialized markup discussed above. In addition, the setup file is used to define a number of options which controls the kind and amount of processing done by the elucidator tool.

The editor that supports elucidative programming is aware of all files in a documentation bundle. The editor awareness is used to open, save, process, and close all such files with single operations in the editor. We will describe the editing tool of the elucidative Scheme environment in more details in section 4.2.

3 Example

Before the discussion of the tools in the elucidative Scheme programming environment we will present a concrete example of an elucidative Scheme program, and we will explain the process of its development. The example is intended to illustrate the concepts introduced in the previous section. However, the reader should be aware that the example is too small to illustrate the real needs and challenges of documented program understanding "the elucidative way". Furthermore, we should be aware that elucidative programming is not targeted at program publication in the same way as literate programming, cf. the first requirement in figure 1. Thus, it is not really the intention to polish an elucidative program. In that respect, the example given below may be somewhat misleading.

The elucidator of the example is available at the Internet address

```
http://www.cs.auc.dk/~normark/elucidative-programming/time-conversion/
```

The reader is encouraged to bring the example up in a browser while reading this section of the paper.

The example is concerned with the development of a program that can decode the number of seconds elapsed since January 1, 1970, 00:00:00 to a year, month, day, hour, minute, and second. Most computers can deliver an integer representing this measure of time, and therefore the conversion forms a very useful basis for a convenient and regular handling of time in terms of a number of seconds.

Using the editor tool of the elucidative programming environment we create the documentation bundle and the underlying directory structure, which hosts an empty source file, an initial template of the documentation file, the setup file, a directory for internal files, and a directory for HTML files. This is done by the editor command `make-elucidator`. The editor prompts the user for all necessary information and creates these files and directories automatically. Next the user issues the command `setup-elucidator` which reads the documentation bundle into editor buffers, and establishes the characteristic splited window view on the documentation and the program (still empty, of course). Now the elucidative programming process can start.

First we establish a little context around the problem. We discuss how to possibly attack the problem. Two approaches are identified, and we happen to go for a mixture of them in our solution. We shift between writing a piece of documentation, and writing pieces of programs. In case a name exists in the program it can be smoothly transferred to the documentation buffer. This makes the writing about the program relatively easy and "secure". We run the

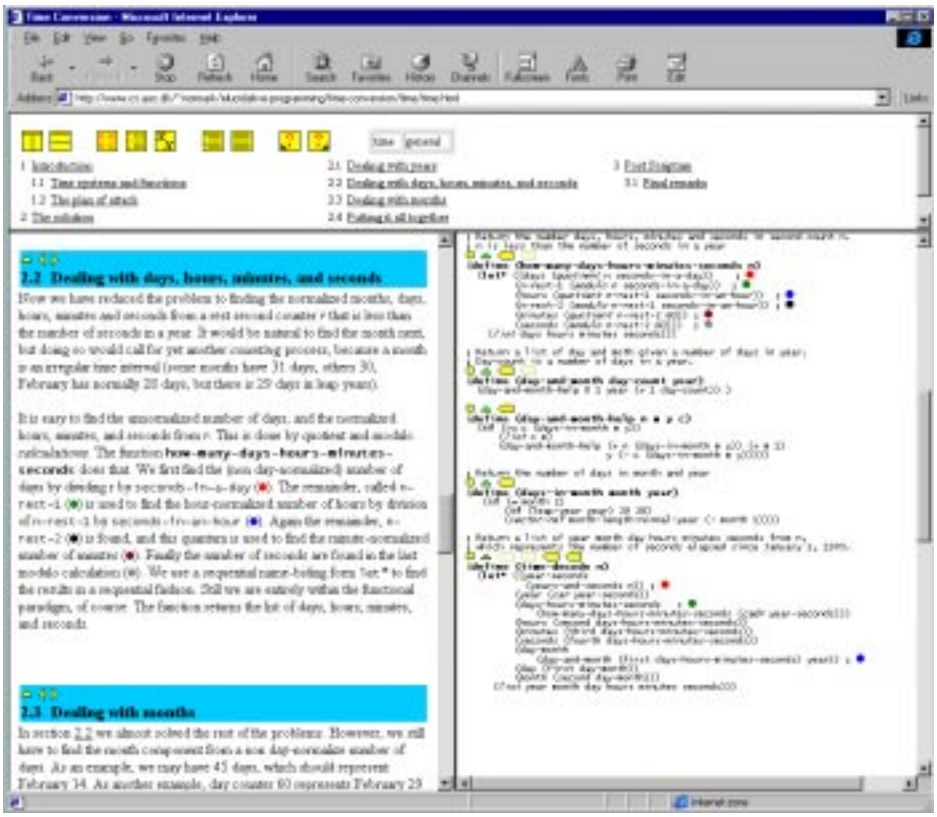

Fig. 4: *A screen shot of the Scheme Elucidator.*

elucidator regularly and refresh the editor in order to get access to a list of known identifiers. We introduce concepts (such as normalization) in order to write about the program in a concise and precise way. This sharpens our understanding of the problem, and makes the solution easier to understand, hereby easing the development of the program. We introduce source markers for program details which we want to address in details in the explanations.

It takes longer time to produce an elucidative program than just to write the Scheme source program. However, it is our firm belief that the quality of the program is improved through this process. Several author's of literate programs support this observation [13, 24]. Furthermore it should be evident that the construction of the documentation is an investment which, to some degree, will pay off when we need to modify the program. Notice, however, that future program modifications implies a substantial work on updating the program understanding, as represented by the documentation.

```
.ENTRY days-hours-minutes-seconds
.TITLE Dealing with days, hours, minutes, and seconds
.BODY
Now we have reduced the problem to finding the normalized months,
days, hours, minutes and seconds from a rest second counter <em>r</em>
that is less than the number of seconds in a year. It would be natural
to find the month next, but doing so would call for yet another
<em>counting</em> process, because a month is an irregular time
interval (some months have 31 days, others 30, February has normally
28 days, but there is 29 days in leap years).<p>

It is easy to find the unnormalized number of days, and the normalized
hours, minutes, and seconds from <em>r</em>. This is done by quotient
and modulo <em>calculations</em>. The function
{*how-many-days-hours-minutes-seconds} does that. We first find the
(non day-normalized) number of days by dividing r by
{seconds-in-a-day} (@a). The remainder, called {-n-rest-1} (@b) is
used to find the hour-normalized number of hours by division of
{-n-rest-1} by {seconds-in-an-hour} (@c).  Again the remainder,
{-n-rest-2} (@d) is found, and this quantum is used to find the
minute-normalized number of minutes (@e). Finally the number of
seconds are found in the last modulo calculation (@f).

We use a sequential name-biding form {-let}* to find the results in a
sequential fashion. Still we are entirely within the functional
paradigm, of course. The function returns the list of days, hours,
minutes, and seconds.
.END
```

Fig. 5: *An excerpt of the documentation source text.*

Figure 4 shows a snapshot of a browser which presents the result produced by the Elucidator.[3] The three frames in the browser correspond to the panes of the basic layout, as illustrated in figure 2. The menu and index pane shows the detailed table of contents of the documentation.

Figure 5 shows a portion of the documentation source, in order to illustrate the specialized markup introduced for our purposes. Here we see the mixture of specialized markup (roff-like 'dot notation' at the start of a line) and HTML markup. The excerpt in this figure corresponds to section 2.2, as shown in figure 4.

4 Tools

There are two important tools in an elucidative programming environment. The most central of these produces the presentation of the program and the documentation. This is the tool we call an *elucidator*. (In some contexts we will also talk about the WWW pages produced by the elucidator, as presented in an Internet browser, as an elucidator). The other tool is the *editor*. It is the qualities of the editor tool that make it realistic and feasible to produce a program and its related documentation. Without specialized editor support, elucidative programming is probably out of reach for most programmers.

In the following two sections we will discuss the Elucidator tool and the editing support of the Scheme Elucidator.

[3] For a better presentation, please consult the on-line version of the example at http://www.cs.auc.dk/~normark/elucidative-programming/time-conversion/

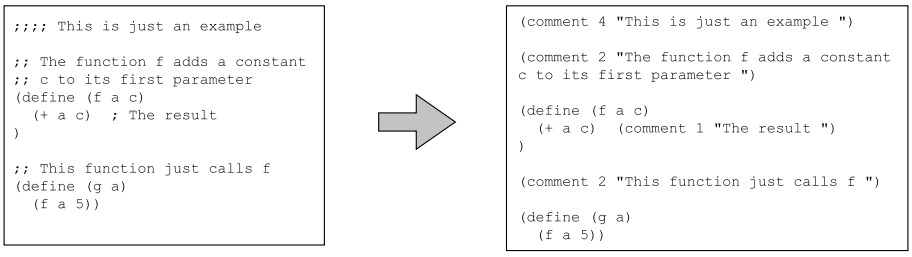

Fig. 6: *An illustration of the pre-processing of comments.*

4.1 The elucidator tool

The elucidator is composed by two major components: an abstractor and a synthesizer. In turn, both the abstractor and the synthesizer has a documentation part and a program part.

The *program abstractor* parses a Scheme program in order to identify the relevant program entities. Similarly, the *documentation abstractor* localizes the documentation entities. Both abstractors are relatively easy to construct due to parser-friendly syntactic basis of Lisp, and due to the simple nature of the documentation markup. The result of the program abstraction process is lists of defined name occurrences tagged with additional information, such as source file belongings. The list is stored in a file in the internal directory of the elucidator. As a processing option it is possible avoid parsing files which are rarely modified; The lists of definitions in such program source files are taken from the internal files produced by an earlier elucidation process.

Sectional comments which identify subsequent sections of Scheme forms cause a number of problem in the parsing process. The reason is that Scheme comments, like comments in most other programming languages, are lexical elements. As such they disappear before a conventional parsing starts. It would, of course, be possible to use a specialized parser which reads the comments. As an alternative, we run the programs through a pre-processor that converts lexical comments to syntactical comments.[4] The result of the pre-processing is stored in temporary files in an internal directory of the elucidator tool. Figure 6 illustrates the source-to-source transformation done by the pre-processor.

The *program synthesizer* decorates the source programs with HTML tags. The HTML `a` tags are the most important because they represent the entity relations and links discussed in section 2, including the anchor points of the source marker. A few icons are added in front of top level definitions. The yellow left arrow icons are the most important because they connect top-level define

[4] We use the pre-processor of the SchemeDoc tool from the LAML software package [17]. Like JavaDoc, SchemeDoc extracts interface comments of definitions. They serve as interface documentation of library collections, and like in Java, they are rendered as HTML files for exposition in an Internet browser.

forms to the places where they are discussed (using *strong doc-prog* relations) or mentioned (using *weak doc-prog* relations) in the documentation. Using these it is easy to identify the places in the documentation where a given Scheme definition is explained. The program synthesizer also decorates the program text with colors and font faces. This decoration is mainly done in order to visualize the *navigational role* of names and other program constructs in the Scheme program.

The program synthesis process is implemented by reading the program source text char by char, while simultaneously traversing the parse tree of the pre-processed program. The information in the parse tree makes it possible to look ahead in the character input stream. As an example, the possibility of looking ahead is crucial for the insertion of destination anchors (`a` name tags) in front the comment which defines the interface of the Scheme functions.

The documentation synthesizer transforms the specialized markup and the source markers to HTML markup. This is implemented by means of a state machine. The state machine applies the knowledge established by the abstractors to target the *doc-prog* links to the program, and the *doc-doc* links to other sections in the documentation. Structural links between sections and entries of the documentation are also inserted in this phase.

Based on the result of the abstraction process the elucidator tools makes a number of useful indexes:

- An index of the definitions in the program
- A cross reference index of the names in the program (only names that are bound at top-level).
- A table of contents of the documentation (in two different depths).

All indexes are presented in the 'Menu and index pane', cf. figure 2. The entries in the indexes are anchors for links to the appropriate entities in the documentation or the program. The cross reference index maps names to all the definitions, in which they are applied. As a convenient shortcut, a definition of (say `N`) in the synthesized program is prefixed with an icon[5] which allows navigation to the name `N` in the cross reference index. Using these shortcuts it is relative easy to follow a selected chain of function calls, from the details towards the overall program structures (upwards in a possible procedure calling chain).

The Scheme Elucidator creates a fixed number of HTML files, which present the files from a documentation bundle. All the bindings are done at "elucidation time". As an alternative, the bindings may occur at a later time, and ultimately at "browse time". Using the latter approach, a program at the WWW server (a CGI program or a Java servlet, for instance) may synthesize the information from a *repository*, which is common for all tools in the elucidative programming environment. The Java Elucidator, which we briefly touch on in section 6, is oriented towards this approach.

[5] The small green triangular icons in front of the top-level definitions brings us to an entry in the cross reference index.

4.2 The editor tool

The editor tool of the elucidative Scheme programming environment presents the documentation and a selected program file in a splited window, in a similar way as the Elucidator. We use a customized version of the Emacs editor. The customization is programmed in Emacs Lisp.

The editor offers navigation possibilities which are similar to the facilities in the browser discussed above. More specifically, the following kinds of navigation are supported via a generic `elucidator-goto` command in the editor:

- navigation from a program name N in the documentation to the definition of N in a program.
- navigation from a defining, top-level name occurrence N in a program to a section in the documentation that explains N.
- navigation from an applied name occurrence N in a program to the corresponding defining name occurrence.
- navigation from one section to another in the documentation via a *doc-doc* cross reference link.

Navigation steps are stacked in order to provide for convenient backing up to previous locations (using the `elucidator-back` edit command). The navigation made possible by `elucidator-goto` and `elucidator-back` is more powerful than plain text searching, because it may move the focus from one Emacs buffer to another. Currently the editor does not support direct navigation between pairs of source markers.

As it can be seen in figure 5 anchored links a represented by specialized markup in the documentation. As an example, {`multiplum-of`} refers to the place of the definition of the function `multiplum-of`. The editor supports the creation of this markup, in particular the entering of names such as '`multiplum-of`'. The name may either be taken and transferred from a program window, or it may be entered by means of Emacs completion (just type the first few letters of a name and Emacs will finalize it). Both of these creations are supported by the editor command `prog-ref`. In a similar way, `doc-ref` supports the creation of cross reference links between sections in the documentation.

The editor shows the documentation in raw and undecorated form, without any special rendering of the elucidator-specific markup nor the HTML markup. Therefore, it is much more pleasant to explore an elucidative program in an Internet browser than in Emacs. In the development situation, it is attractive to use both an editor and a browser. The core navigational functionalities are overlapping. The editor is more flexible with respect to searching than both Netscape and the Internet Explorer. The browser provides more elaborate and more user friendly navigation.

The editor depends on information from the elucidator tool. As explained in section 4.1 the elucidator saves the information, which is extracted by the abstractors. This information is used by the editor to support both navigation and flexible creation of links. When the elucidator finishes its processing, the

editor command `refresh-elucidator` updates the editor's knowledge about the documentation bundle.

The editor offers a number of other convenient commands specific to the Scheme elucidator. The editor knows the files of a documentation bundle. At any given point in time, one of the program files is in focus. The command `show-program` brings another program in focus. The command `reset-elucidator` establishes a split-window, with documentation in the upper part and a program in the lower part. The `reset-elucidator` command is very useful if Emacs has been used for other and perhaps non-related purposes, such as mail reading or plain file editing.

We find that the use of Emacs is a better alternative than proposing a new and special editing tools, targeted exclusively at the creation of elucidative programs. It seems to be a general experience that programmers are reluctant to use brand new program construction tools.

In the ideal situation, however, the elucidative editing tool should be part of an existing integrated development environment. In that situation, support of elucidative programming would be implanted into an existing and more complete environment, which hereby would be *documented enabled*.

5 Related Work

The field of literate programming is the foundation of our work on Elucidative programming. Literate programming was coined by Knuth in 1984 [13] as a result of major software undertakings with the TeX text formatting system [14]. Knuth's research group implemented the WEB system [12, 15] as a set of tools (weave and tangle) which supports literate programming. After that a number of similar systems appeared [2, 31, 10]. The main variations stem from the programming and documentation languages supported. Some systems, such as Noweb and Nuweb are programming language independent. There exists a published annotated bibliography of literate programming [27]. However, the most complete and up-to-date bibliography is available via Nelson H. F. Beebe's home page on the Internet.[6] In our earlier paper on elucidative programming [20] we refer to a number of small examples of literate programs (mainly from *The Communications of the ACM* in the late eighties) and to other literate programming approaches than WEB-systems.

In the Scheme world there exists a couple of systems, Scheme WEB and SLaTeX, that somehow represent a bridge to the world of literate programming. See the Scheme repository [32] for details on these.

Sametinger and colleagues at Johannes Kepler University of Linz, Austria, have developed the DOGMA programming environment for C++ [26]. The DOGMA project has in the same way as the elucidative programming project gained inspiration from literate programming. Both systems are based on relations between documentation units and program units. The relations are used

[6] `http://www.math.utah.edu/pub/tex/bib/litprog.html`.

to make relevant documentation appear when documented program units are in focus. It is worth mentioning that Samtinger et al. have developed a notion of *object-oriented documentation* for DOGMA [25]. Using these ideas, object-oriented concepts (such as inheritance) is used on sections of documentation, which are attached to classes in C++. DOGMA is a complete and integrated programming environment for C++, including a residential text editor. As a contrast, the elucidative programming environments that we have developed consist of two separate and partly overlapping tools (an editor and a browser, which is powered by the elucidator) with a relatively simple integration in between them. In particular, we rely on an external editor (Emacs) which is customized to support an elucidative programming process. This may seem to be of minor importance; Nevertheless, we believe that most programmers are conservative in adopting new editing environments, and as such our less integrated environment may turn out to be a good, pragmatic alternative to a system like DOGMA.

In our current work on elucidative programming we use the World Wide Web and the Internet as a medium for program documentation. JavaDoc [3] is the main inspiration with respect Internet mediated program documentation. JavaDoc makes it possible to extract interface documentation from comments in Java programs that are marked in a special way and follow special conventions. The extracted interface documentation is organized as a set of interrelated WWW pages. JavaDoc documentation is particularly useful for documentation of class libraries, and as such it is oriented towards program reusability. Elucidative program documentation is targeted at the team of programmers who are responsible for further development and maintenance of particular applications.

Work on program understanding can be categorized in at least two different groups: *Prevenient* and *posterior* approaches. Our work represents the prevenient approach. As discussed in this paper the idea is to document the program understanding before, or side by side with the development of the program. We hypothesize that interleaved documentation and program develop processes improve the quality of our programs. Furthermore, we see the documented program understanding as an investment that pays off during the maintenance phase. However, we are aware that the prevenient approach is somewhat idealistic, and that mainstream program development takes place without much emphasis on pro-active documentation of the program understanding. The posterior approach deals with extraction of program understanding from existing programs. A variety of different "reverse engineering" tools have been proposed for such endeavors (for an overview, see [30]). Because of the relative dominance of the posterior approach there is much literature on this branch of program understanding. The IEEE *International Workshop on Program Comprehension* is the main forum for reverse engineering papers [4–9].

A few years ago our work was focussed on *hyper structured programming environments*. We carried out a number experiments with a system called HyperPro. The main contribution of this work was the notion of rich hypertext, and in particular the possibilities of defining flexible interaction schemes on rich hypertexts [21, 23]. The work on elucidative programming steps away from a fine

grained hypertextual representation of the underlying programs and documentation. In our current tools we deal with traditional and coarse grained program source files. As such, our current work is much more pragmatic than the work on HyperPro.

Kasper Østerbye's work on literate Smalltalk programming [22] was an important part of the HyperPro project. Like in HyperPro, Østerbye's Smalltalk environment was based on a fine grained representation of classes, methods, and textual documentation. The work by Reenskaug and Skaar [24] is also about literate programming support for Smalltalk.

It is natural to study the needs for program documentation in relation to both the analysis and design phases of the software development process. UML is the dominating 'language' for description of analysis and design artifacts. The work by Vestdam et al. (from our group) describes a contribution to a CASE system which introduces the idea of documentation threads [29]. Documentation threads may involve elements from UML diagrams, program fragments, as well as pieces of documentation.

6 Status and conclusions

We have described the ideas and concept of an elucidator tool for Scheme. In addition, we have outlined the facilities of the editor that supports the creation and modification of elucidative Scheme programs.

The idea of dividing a window (or screen) in a documentation pane and a program pane, in between which mutual navigation takes place, is central to the elucidator. A complicated programming situation is often characterized by juggling with many aspects of the program at the same time. Often it is difficult and demanding to mobilize sufficient concentration on all these aspects (pieces of the program). In this situation the documentation pane may be used to keep a number of program parts together (by means of links) in a way, which makes it easier and safer to handle a complex programming task.

The Scheme Elucidator is in local use at Aalborg University. The most substantial elucidative program documents the development of the tool itself. This program can be seen on the *elucidative programming home page* together with a number of smaller examples [16].

A group of master thesis students at Aalborg University have developed an elucidator for Java [28]. 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 the abstracting processes in a relational database (whereas the Scheme environment uses lists of names stored in files). Furthermore, the Java elucidator generates the HTML pages by demand on the WWW server (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* referred above.

The work on the Scheme and Java elucidators raises several interesting questions. First, is it possible to convince and discipline programmers to document their program understanding? Second, is it possible to convince the managers of program development projects to invest in an improved program quality, by means of documented program understanding "the elucidative way". Third, can the documented program understanding of an elucidative program be maintained with reasonable means? And finally, can we develop practical documentation patterns that will allow average programmers to write good elucidative documentation of their programs? In the next couple of years we hope to find good answers to these questions.

The Scheme Elucidator is available as free software from the LAML home page on the Internet [17].

References

1. Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 482–498. ACM, May 1993.
2. Preston Briggs. Nuweb, A simple literate programming tool. Technical report, Rice University, Houston, TX, USA, 1993.
3. Lisa Friendly. The design of distributed hyperlinked programming documentation. In Sylvain Frass, Franca Garzotto, Toms Isakowitz, Jocelyne Nanard, and Marc Nanard, editors, *Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France*, 1995.
4. IEEE. *Proceedings of the Second Workshop on Program Comprehension*, July 1993.
5. IEEE. *Proceedings of the third Workshop on Program Comprehension - WPC'94*. IEEE Computer Society Press, November 1994.
6. IEEE. *Proceedings of the fourth Workshop on Program Comprehension*. IEEE Computer Society Press, March 1996.
7. IEEE. *Fifth International Workshop on Program Comprehension*. IEEE Computer Society Press, March 1997.
8. IEEE. *Sixth International Workshop on Program Comprehension*. IEEE Computer Society Press, June 1998.
9. IEEE. *Seventh International Workshop on Program Compresension*. IEEE Computer Society Press, May 1999.
10. Andrew L. Johnson and Brad C. Johnson. Literate programming using `noweb`. *Linux Journal*, 42:64–69, October 1997.
11. Richard Kelsey, William Clinger, and Jonathan Rees (editors). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.
12. Donald E. Knuth. The WEB system of structured documentation. Technical Report STAN-CS-83-980, Department of Computer Science, Stanford University, September 1983.
13. Donald E. Knuth. Literate programming. *The Computer Journal*, May 1984.
14. Donald E. Knuth. *The Texbook*. Addison-Wesley Publishing Company, 1984.

15. Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0.* Addison Wesley, 1993.
16. Kurt Nørmark. The elucidative programming home page. http://www.cs.auc.dk/-~normark/elucidative-programming/, 1999.
17. Kurt Nørmark. The LAML home page. http://www.cs.auc.dk/~normark/laml/, 1999.
18. Kurt Nørmark. Programming World Wide Web Pages in Scheme. *Sigplan Notices*, 34(12):37–46, December 1999. Also available via [17].
19. Kurt Nørmark. Using Lisp as a markup language—the LAML approach. In *European Lisp User Group Meeting*. Franz Inc., 1999. Available via [17].
20. Kurt Nørmark. Requirements for an elucidative programming environment. In *Eight International Workshop on Program Compresension*. IEEE, june 2000. Can be found via [16].
21. Kurt Nørmark and Kasper Østerbye. Rich hypertext: A foundation for improved interaction techniques. *International Journal of Human-Computer Studies*, (43):301–321, 1995.
22. K. Østerbye. Literate Smalltalk programming using hypertext. *IEEE Transactions on Software Engineering*, 21(2):138–145, February 1995.
23. Kasper Østerbye and Kurt Nørmark. An interaction engine for rich hypertexts. In *European Conference on Hypermedia Technology 1994 Proceedings*, pages 167–176. ACM Press, September 1994.
24. Trygve Reenskaug and Anne Lise Skaar. An environment for literate Smalltalk programming. *Sigplan Notices*, 24(10):337–345, October 1989.
25. J. Samtinger. Object-oriented documentation. *Journal of Computer Documentation*, 18(1):3–14, January 1994.
26. J. Samtinger and S. Schiffer. Design and implementation aspects of an experimental C++ programming environment. *Software Practice and Experience*, 25(2):111–128, February 1995.
27. L. M. C. Smith and M. H. Samadzadeh. An annotated bibliography of literate programming. *Sigplan Notices*, 26(1):14–20, January 1991.
28. Søren Staun-Pedersen, Max R. Andersen, Vathanan Kumar, Kristian L. Sørensen, and Claus N. Christensen. The elucidator - for Java. Preliminary master thesis report, Janunary 2000. Available from http://dopu.cs.auc.dk.
29. Thomas Vestdam. Pulling threads through documentation. In Mughal and Opdahl, editors, *Proceedings of NWPER'2000 - Nordic Workshop on Programming Environment Research*, May 2000.
30. Richard C. Waters and Elliot Chikofsky. Reverse engineering: Progress along many dimensions. *Communications of the ACM*, 37(5):22–25, May 1994.
31. Ross Williams. FunnelWeb user's manual. Technical report, University of Adelaide, Adelaide, South Australia, Australia, 1992.
32. John Zuckerman. The internet Scheme repository. http://www.cs.indiana.edu/-scheme-repository/home.html.