The CLiP Style of Literate Programming

Eric. W. van Ammers and Mark R. Kramer

Computer Science Department
Wageningen Agricultural University
Dreijenplein 2, 6703 HB Wageningen, The Netherlands
voice: +31 (0)8370 83356, fax: +31 (0)8370 84731
e-mail: ammers@rcl.wau.nl

Abstract:

Literate programming is a method to integrate programs with their documentation. Compilable modules are no separate entities, but they are extracted from the documentation.

Most literate programming tools use explicit commands in the formatter input files to guide the extraction process. The CLiP approach presented in this paper relies on coding style instead. This makes the code extraction completely independent of the text processing environment. Thus CLiP can even be used in combination with a modern wordprocessor.

In addition the CLiP mechanism is independent of programming languages as well, and the CLiP style is easily adapted to any language.

Keywords:

Literate programming, Language independent literate programming, Text processing independent literate programming, Multi-file/multi-module literate programming, Documentation, Coding style.

1 Introduction

Maintenance is the dominating cost factor of a software system. Estimations are that maintenance programmers spend up to 50% of their time simply trying to *understand* what the code does [Parikh and Zvegintov 1983]. Also it has been observed over and over that conventional documentation is often inadequate for maintenance purposes. Programmers on a maintenance job tend to ignore the documentation and rely on the code listings instead. The fact that in a conventional set-up the documentation and the code are distinct documents, is generally recognized as a contributing factor since it presents a serious obstacle to keep the code consistent with its description. The problem made Brooks ponder on the idea of "self-documenting code", that is an improved documentation method which integrates high quality documentation with actual code [Brooks 1982].

Literate programming promises a significant step towards more comprehensible programs [Knuth 1984]. The basic idea is twofold. Firstly the program code and its description are integrated in such a way that compilable units can be extracted automatically, thus implementing the ideal of self-documenting code in an even more extreme form. Secondly the integrated document is no longer presented as a plain ASCII-file, but rather as a type-set document where all sorts of (typo)graphical features can be applied to increase its explanatory power.

Evidently literate programming requires a mechanism to create compilable units — henceforth called *modules* — from the documentation files. Such a mechanism is called a module extractor. In most literate programming settings the documentation is produced be means of a formatter. In that case the literate programmer has in fact to work with four different languages simultaneously, namely the explanation language (typically English), the programming language (e.g. Pascal), the language to instruct the formatter and the language to command the module extractor [Reenskaug 1989].

The CLiP approach distinguishes itself from other literate programming environments in that it applies *style* (or programming conventions) rather than *commands* to conduct the module extraction process. This concept sets the programmer entirely free to use any text processing system he prefers. In fact CLiP cooperates just as easily with a batch oriented formatter as with an interactive word-processor. A second important quality of CLiP is that it processes virtually any programming language. However, in this respect it is not unique.

The rest of this paper consists of five main parts. In section 2 we discuss the literate programming paradigm. We then continue with an exposition of the CLiP approach which we illustrate by an example in the appendices. In section 4 the CLiP technique is compared to other literate programming tools around. Section 5 summarizes the experiences that have been gained so far. Finally we reflect on some important developments of literate programming that we foresee.

2 The literate programming paradigm

Knuth's original paper describes literate programming as [Knuth 1984]

> "Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do".

We would like to add the phrase "in such a way that it can be interpreted by a computer" and consider this an almost perfect definition.

Thus a literate program contains the *actual code* (the lines that end-up in the modules) divided in suitable chunks. In general each of those chunks will be accompanied with a *description* (an explanation in a suitable form). The author integrates the two in a way that in his opinion provides the best explanation of the program. This integrated text we will call the *documentation* of the system. To have a computer interpret the code, the documentation has to be processed to realise modules. In this section we will explore the ingredients to achieve this objective.

2.1 Explanation to human beings

Programming is a painful process involving many distinct steps. With every step we associate a design decision which in turn is implemented by a certain amount of ’real’ code. A literate program documents the relation between code and design decision as an integrated entity. In this way the ideal of self-documenting code is closely approximated. As an additional advantage the programmer tends to express the design step in terms of actual names as they show in the program code. In a conventional environment, where description and program code are distinct documents, the tendency is rather to express descriptions in more abstract terms and names that often do not return as identifiers in the program.

Design steps per se are fairly autonomous entities. It is by their mutual relationship that a certain desired functionality is realised. Therefore the hierarchy of the design steps is a crucial factor to understand the structure of a software system. This hierarchy usually does not reflect the chronological order of the design decisions that have been made [Parnas 1986]. Literate programming invites the programmer to explain the hierarchy in any order he considers appropriate and in terms of the code that is actually involved. Whether this exposition is top-down, bottom-up, middle-out or any other suitable way is entirely up to the author.

The freedom to present the design hierarchy in the order desired by the programmer/author is considered a vital aspect of the literate programming paradigm [Bentley 1986a/b]. Knuth appreciates this property in that it "... allows a person to express programs in a ’stream of consciousness’ order." [Knuth 1984].

Programming is not a rational design process [Parnas 1986]. Design steps are proposed, explored, thrown away and replaced by alternatives. For this reason it is customary in a conventional environment to produce the description of a program only after its design has stabilized. With literate programming the work process is much more flexible. Since the code and its description are one and the same document it is easy to jot down any noteworthy remarks together with the related code. Whether or not these remarks will mature into actual descriptions, depends mainly on the way the design stabilizes. But the remarks as such are secured and there is no chance they will be accidentally forgotten in the final documentation.

A literate program can (and should) be organized as a textbook and a notorious example is the TEX-program by Knuth [Knuth 1986]. Thus, for the sake of explaining the documented program, we can make use of all those features that are used for ordinary textbooks too. The benefits of the so called *book format paradigm* are described by Oman and Cook, who also conducted experimental studies to verify those claims [Oman 1990a/1990b]. It turns out that organizing information as components of a book (i.e. preface, table of contents, indices and pagination, chapters, sections, paragraphs, sentences, punctuation, type style, character case etc.) provides a variety of access methods which have a significant impact on program comprehension. Also facilities like figures, schemes, drawings, tables, etc. greatly extend the scale of gadgets that can be thrown in for explanatory proposes.

2.2 Interpretation by a computer

The relationship between design decisions becomes manifest when a design step is expressed in terms of design steps described elsewhere in the literate program. Of course conventional references to names are solved by the language environment in question. But in a literate programming environment the order of the code fragments has to be indicated as well. This is accomplished by placeholders that refer to other code fragments in the documentation. Typically, a pseudostatement is a placeholder for the code fragment of the supplemental design step. At module-generation-time a literate programming system will *automatically* expand every placeholder into the intended code described elsewhere in the documentation. Since the module generation process is not disturbed by human intervention, the modules necessarily reflect the code as it appears in the documentation. This property of literate programs has been dubbed *verisimilitude* [Van Wyk 1990].

The documentation of a computer program in general extends over many different files. A program itself is seldom monolithic; virtually always it is composed of several independently compilable modules. Sometimes it may be convenient to be able to derive several versions of a program (e.g. a test version containing additional debugging code or different versions for different platforms). For these reasons it is highly desirable that a literate programming environment does not impose restrictions on the way the modules relate to the documentation. Thus a literate programming tool should be able to extract an arbitrary set of program modules from an arbitrary set of documentation files.

3 The CLiP approach to literate programming

In the CLiP approach the preparation of printed documentation is totally decoupled from the code extraction. The one and only function of the tool CLiP itself is to extract the code from the documentation; CLiP stands for "Code from Literate Program".

CLiP can be used in combination with any formatting tool and/or word-processor, as long as plain text files are available as source files for the module extraction. With ’traditional’ formatters like troff or (La)TeX the input files of the formatter can be used (see fig. 1a). To use a modern word-processor, the CLiP source files have to be generated by a suitable ASCII-export from the word-processor (see fig. 1b).

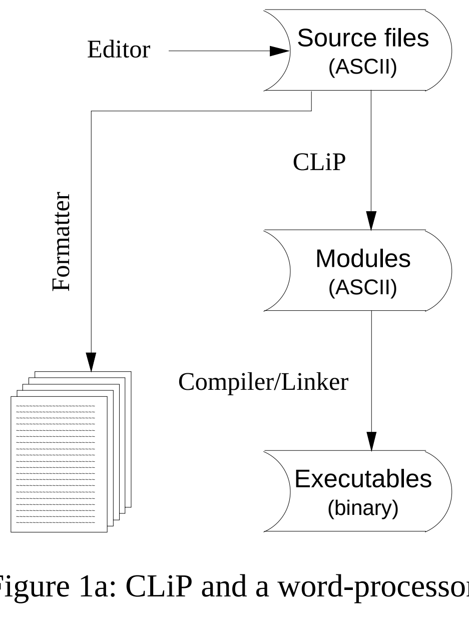

Figure 1a: CLiP and a word-processor

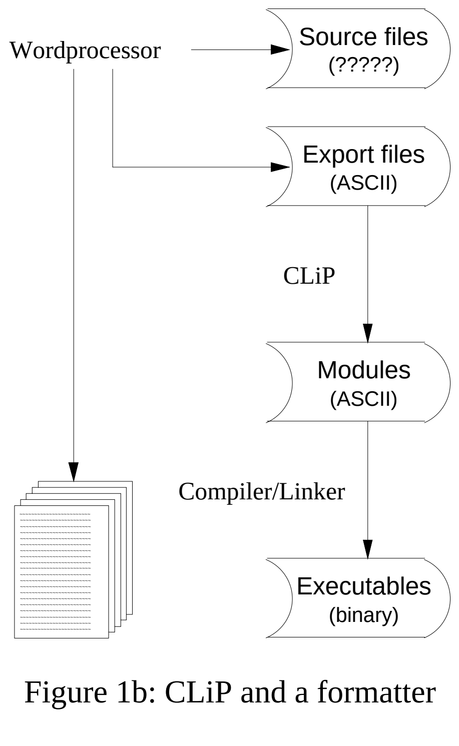

Figure 1b: CLiP and a formatter

The independence of text processing and CLiPping was one of our initial requirements. Other requirements included a total independence of target (programming) language and the freedom to describe several modules in one document or a set of related documents. We return to these ideas in section 3.3. Last but not least we wanted the code contained in the documentation to look natural. More specifically, it should not contain special commands for the code extractor.

The placeholders introduced in the process of literate programming must look natural, yet have to be distinguished from ’normal’ code. Placeholders may be used to represent pseudo-statements, but also to indicate locations where definitions, declarations, externals etc. have to be included. In fact the nature of the ’code’ to be inserted does not really matter: even data files may be organised in this way. Usually, in algorithm development the placeholders (i.e. pseudocode) are represented in the form of comments inside fragments of code. CLiP adopts this convention for all other placeholders as well. Thus in the CLiP approach all information relevant to the code extraction is represented by a naturally looking coding style, viz. comments that resemble pseudo-statements.

This representation is expounded in the following subsections: first we look at the bare syntax of CLiP, next some additional conventions are presented. In section 3.3 we investigate some advanced ways of using CLiP. The appendices A and B contain a small, yet complete, example of a literate program in CLiP style which is augmented with line numbers. The numbers between "<" and ">" in the discussion below refer to these lines.

3.1 Basic syntax of CLiP

In the structure of a CLiP-document, as far as module extraction is concerned, we may distinguish four syntax levels:

1. division in active parts with a special meaning to CLiP (code fragments) and passive parts which CLiP ignores (descriptions),
2. structure of the active parts: the segments,
3. internal structure of segments,
4. the concrete syntax.

The first three levels constitute the abstract syntax in which the basic entities are complete lines. At the fourth level the internal structure of these lines is taken into account.

The purpose of the extraction tool is to process only those lines that ’look like’ code or programming language comments (all numbered lines). To keep the parser independent of programming languages, only comment lines with a special appearance trigger the process of CLiPping. We call these lines CLiP-lines <e.g. 1, 7, 8, 16, 17, 20, 26-29, 32>.

At the first level of syntax we distinguish descriptions (all unnumbered lines) and code fragments, in the terminology of CLiP called ’stubs’ (e.g. 1-7, 8-32, 37-38, 39-41, 49-75):

Source-file == Description (Stub Description)*

At the second level the internal structure of Stubs is introduced. Each stub starts with a segment of CLiP-lines stating the name (and optionally some properties) of the stub <typically 49-51, 86-88>. The stub ends with a segment of CLiP-lines containing a special ’end string’ <41, 75, 109>. Any segment of CLiP-lines in between constitutes a Slot-segment <54-57, 61-64, 68-70, 105-107>. Note that ’slot’ is the CLiP terminology for a placeholder. All non-CLiP-lines in a stub establish the Code-segments <52-53, 58-60, 65-67, 71-74, 89-104, 108>:

Stub == Stub-segment (Code-segment | Slot-segment)* End-segment

Although descriptions do not have an internal structure, we introduce the notion of a Description-segment for completeness sake:

```
Description == Description-segment
```

The Stub-segments, Slot-segments and End-segments consist entirely of consecutive CLiP-lines. Each one starts with a CLiP-line that contains additional characters <49, 54, 61> to distinguish it from ordinary comments <10, 11, 12, 117, 123> as well as from other CLiP-lines <50, 51, 55, 56>. Such starting lines, named L1 and L2, basically have the same structure. The distinction is made according to the contents of the lines. Additional lines of one of these segments should belong to category L3 <27, 28, 50, 51>. Code-segments and Description-segments are made up of non-CLiP-lines, called L4 here <18, 19, 52, 53, 137, 138, 139>:

```
Stub-segment ==            L1 L3*
Slot-segment ==            L1 L3*
End-segment ==             L2 L3*
Code-segment ==            L4*
Description-segment ==     L4*
```

Note that Stub-segments and Slot-segments have the same syntax. They are distinguished by the fact that the first segment of a stub is always a Stub-segment while all others are Slot-segments.

The appearance of CLiP-lines is largely free to the user. They start and end with sequences comm-start and comm-end intended to delimit comment lines of the programming language at hand ("(*" and "*)" in our example). To distinguish CLiP-lines from other comment lines these sequences are extended with one ore more special characters, the CLiP-char (in our case "*") . The number of CLiP-chars constitutes the difference between L1 and L2 on the one hand (several additional CLiP-chars <49, 54, 75>) and L3 on the other hand (exactly one additional CLiP-char <50, 51, 55, 56>). In practice we emphasize the distinction by using a longer row of CLiP-chars.

```
L1 == comm-start (CLiP-char)^2+ text (CLiP-char)^2+ comm-end
L2 == comm-start (CLiP-char)^2+ end-string text (CLiP-char)^2+ comm-end
L3 == comm-start CLiP-char text CLiP-char comm-end
L4 == {anything else including ordinary comments}
```

The text of a complete stub segment or slot segment effectuates the name of the stub or slot. These names are used by CLiP to match stubs to their corresponding slots. In addition a stub or slot may contain a number of options following the name. Each option is introduced by a special option-marker <16, 17, 106>.

The actual representations of the various syntactic categories are run-time parameters of CLiP. For Pascal we use

```
comm-start ==       "(*"
comm-end ==         "*)"
CLiP-char ==        "*"
end-string ==       "End of"
option-marker ==    "#"
```

3.2 Conventions

The syntax presented above is a good starting point for an intuitive presentation of literate programs. Yet, it gives the opportunity to augment the notation with a specialized project style. Our experience indicates that some additional conventions are even necessary to reach our aim of 'explaining the code to humans' and 'almost incidentally explaining it to the computer as well'.

In this section we restrict ourselves to the conventions used in the example of the appendices. The conventions partly deal with rules for organising the documentation and its appearance in the type-set form, which aspects are independent of CLiP. Partly they elaborate the use of the CLiP syntax.

In the type-set documentation all code should appear in a non-proportional font, to aid navigation in module listings during debugging. In this way all code appears in the modules exactly as it shows up in the documentation. Furthermore all identifiers that are mentioned in the descriptions should be in the same font as the code fragments (maybe in another size).

Sections of the documentation should reflect single design steps with all corresponding code included. In particular, variables introduced in the code of a section should be explained in the description of that section, and the actual declarations should be included in the same section as well. To reduce the visual overhead, a stub can contain a special option "quick", which instructs CLiP that the stub is ended by an empty line or a CLiP-line (this complication is not dealt with in section 3.1. Examples are <33-34, 35-36, 37-38, 42-43, 44-46>).

There are essentially two kinds of slots: pseudostatements and other slots. Pseudostatements are important for the explanation of an algorithms, so pseudostatements should catch the eye <26-29, 54-57, 61-64>. Most other slots serve as ’syntactic’ placeholders to indicate the positions of (e.g.) declarations. Therefore they are kept to one single line with a short slot name <16, 17, 20>.

We write pseudo-statements as framed comments with a short identification in the top line. This identification is not strictly needed, but it aids in the process of finding relations between slots and stubs. The remainder of a slot describes the operation of the pseudostatement in terms of the variables involved. Thus the corresponding stub may be understood independently, essentially avoiding Thimbleby’s ’dynamic binding problem’ [Thimbleby 1986].

3.3 Advanced features

The CLiP tool is able to combine multiple source files. This allows structuring the explanation of a program into separate levels of detail. Each level of understanding abstracts from the details of a lower level. Therefore, in the CLiP approach it is customary to split the documentation of one module into separate documents according to this hierarchy.

Conversely, it is also possible to document several modules in one (set of) source file(s). This feature is typically used to include data files in the documentation: e.g. test data, tables and help files. In the case of Modula-2 programs this mechanism proves very useful too for generating specification and implementation modules from the same sources file. Thus the procedure and function headings need to be specified only once.

The operation of CLiP may be tuned by means of options in slot or stub segments. The default mechanism is designed such that options are not normally needed. The remainder of this section discusses the most important options in the context of their most frequent application.

Declaration slots: options "multiple" and "leader"

The default mechanism of CLiP relates each placeholder to exactly one code fragment. This is a useful choice for placeholders representing pseudocode, but not for ’syntactic’ placeholders, such as a fixed location where variable declarations have to be included. It is very illogical to declare all variables at once, where the actual code is split into manageable parts. Therefore the option "multiple" is used to indicate a slot where more than one stub can be included <16, 17, 20>.

Another peculiarity of declarations is that some languages (notably Pascal) require that each type of declaration is introduced by its own keyword. This leads to problems when the keyword is either included in the first corresponding stub (which might be replaced during maintenance) or in front of the slot (if no stub is specified, the program is syntactically incorrect). To deal with these problems, CLiP allows the "leader" option in a stub <33, 35>. A leader stub is included only if at least one regular stub is present in the documentation; otherwise the leader stub is ignored.

Data files: option "comment off"

Comments are normally extracted with the code. This is important to guide the navigation in the extracted code, especially in the process of debugging (see also section 5). But comments are not desired in data files. With the option "Comment off" we direct CLiP to extract only the code segments <1, 143>.

It is also possible to suppress comments locally by specifying "Comment off" in a slot. This is very useful in a list of variable declarations: all declarations are included without the (superfluous) comment stating the slot-name.

Testing and debugging code: option "optional"

In the process of debugging one often needs to include code for the purpose of tracing or dumping tables. According to the literate programming credo we do not want to edit the generated modules, or even the source files, for this purpose. So we include slots that normally will be left blank, indicated by the option "optional" <105-106>. This option instructs the CLiP tool not to complain if the slot is not satisfied. Those slots serve as placeholders for debugging code that is kept in separate documents (e.g. appendix B). Whether or not the debugging code is included depends solely on the set of source files that is supplied for a run of CLiP.

Partial implementations: option "default"

It is very useful to test each level of a program without all details filled in. When the literate program is structured as explained in the first paragraph of this section, a level of abstraction corresponds to a set of source files. In order to test such a level, the source files should contain some default actions to be taken instead of those missing details, e.g. writing a message "function ... not yet implemented". This code is included in a stub with the option "default" <42-43>, which is only extracted when no regular stub is supplied for the corresponding slot.

A related situation occurs in lines 42-43 of the example where we have a so called *partially defined* data type `TEXT_LINE`. First a temporary declaration is given, adequate to the current abstraction level. Later on, in section A.4, it is refined to its final form.

Other applications of the "default" option

The "default" option is very useful too during the development of multiple versions of a program. In its simplest form it is almost identical to the previous example. Operations that are not allowed in some versions are separated into different source files; a default stub contains code to generate a message like "operation ... not allowed in this version".

Default stubs may also be used for porting and tuning. In this case the default stub gives a general solution, but not necessarily an efficient one. Well tuned (but machine specific) code may be included by use of an additional source file. As a premium the main document is not clobbered with all the tricks that are sometimes needed to gain the desired efficiency.

4 Comparison with other literate programming tools

The most widely known system for literate programming is without doubt Knuth's WEB [Knuth 1984, Bentley and Knuth 1986a, 1986b, Sewell 1989]. WEB consists of two programs, WEAVE and TANGLE, that generate a TeX-file and a Pascal-module respectively. By now WEB has been adapted to a number of other programming languages. CWEB [Levy 1987] for c, MWEB [Sewell 1987] for Modula-2, FWEB [Avenarius and Oppermann 1990] for Fortran and various other WEBs. With Ramsey's Spider [Ramsey 1989] it is even possible to generate an XWEB for your favourite language X. Apart from the target programming language, all these WEBs share the properties of WEB; therefore we refer to them collectively as the WEB family.

In table 1 we compare CLiP with those other tools of which we know sufficient details: The WEB family, Thimbleby's Cweb [Thimbleby 1986], c-no-web [Fox 1990], HSD [Tung 1989], LIPED [Bishop 1992], and a nameless Smalltalk environment [Reenskaug 1989]. We ignore VAMP, the predecessor of CLiP [Ammers 1992].
[note to the referee: possibly we missed some important other tools; we plan to expand this table in the final version].

Most tools are coupled to one specific programming language (row 1). As mentioned in the first paragraph, each member of the WEB family is tailored to one particular language. LIPED is independent of programming language in the sense that it is controlled by a language description. CLiP's mechanism is totally independent of the target language, so e.g. job control procedures and data files (row 2) can be extracted as well. The only other system addressing data files is the Smalltalk environment.

The majority of the literate programming tools is coupled to one or a few batch formatters (row 3). The Smalltalk environment and LIPED have their own text processing interfaces. LIPED may be interfaced to an arbitrary text formatter by means of a trick: it uses printer configuration files which can be used to generate formatter input (e.g. LaTeX-files). CLiP is the only tool suitable for use with a word-processor (see fig. 1b) because all information for the code extractor is visible in the documentation. Alternatively its source files may be edited by any text editor (row 4). The Smalltalk environment, LIPED and HSD have integrated special purpose editors which add functions for navigating in the literate programs as on line documentation (row 5).

CLiP is quite unique in extracting multiple modules from multiple source files (rows 6 and 7). Perhaps LIPED shares this feature, but this is not entirely clear from the literature. The Smalltalk environment interfaces to the Smalltalk library for storing and retrieving all pieces of information.

Suppression of details (row 8) in very easy in CLiP by its ability to process multiple source files. By the same mechanism CLiP is able to handle debugging code (row 9) and to integrate multiple versions (row 10) of the same program (see section 3.3). Debugging code is handled in the WEB family and in cweb by macro's that expand either to comment delimiters (effectively 'commenting out' the debugging code) or to whitespace at the users choice. WEB uses 'change files' for various purposes, one of them versioning (for more details see e.g. Sewell [Sewell 1989] or Appelt and Horn [1986]). We use the term 'versioning' (row 10) because neither CLiP nor WEB do perform real 'version management'. LIPED includes version information in the names of code fragments. The Smalltalk environment relies on the underlying system for these purposes.

	CLiP	...WEB	cweb	c-no-web	HSD	LIPED	Smalltalk-env
1 Programming language	any	see text	c	c	c	several	Smalltalk
2 Data files etc.	yes	no	no	no	no	no	yes
3 Text processing env.	any	TeX	troff	(La)TeX	LaTeX	see text	special
4 Editing	editor / wordproc	any editor	any editor	any editor	built-in editor	built-in editor	built-in editor
5 On line documentation	no	no	no	no	yes	yes	yes
6 Nr. of source files	many	1	1	1	1	1?	(library)
7 Nr. of modules	many	1	1	1	1	1?	(library)
8 Suppressing details	yes	no	no	no	no	no	no
9 Debugging code	yes	yes	yes	no	no	no	yes
10 Versioning	yes	yes	no	no	no	yes	yes
11 Documentation structure	free	limited	free	fixed	fixed	free	free
12 Code formatting	no	yes	no	no	yes	yes	yes
13 Automatic index	no	yes	yes?	no	no	(yes)	(yes)

Table 1: Properties of various literate programming tools

The last part of the table relates to the ’book format paradigm’. All tools allow automatic generation of a table of contents, use of type faces, semi-automatic indexing etc. Indexing of program identifiers is fully automatic (row 11) in the WEB family. The Smalltalk environment and LIPED use an internal representation that allows automated interfacing to the semi-automatic mechanism of a text formatter. With CLiP indexing fully depends on the text processing tools used; no automatic means are available to generate cross reference lists in the type-set documentation. Neither does the CLiP approach support formatting of the code (row 12) in the documentation, but this is felt as an advantage rather than a disadvantage (see section 5).

In most systems, including CLiP, the literate programmer is totally free to choose a suitable structure of documentation (row 13). In WEB, however, only two levels of sections are allowed. HSD generates the documentation as a preorder traversal of the tree of code fragments. Because the source files of c-no-web are the c modules with embedded formatting commands, the order of the code is (necessarily) retained in the printed documentation.

5 Experiences

Many authors have reported their experiences with literate programming in general [Knuth 1984, Thimbleby 1986, Van Wyk 1987, Reenskaug 1989, Oman 1990b, Ramsey 1991, Ammers 1992, Smith 1992, Levy 1993]. Here we report on our experiences with the CLiP approach in particular.

The CLiP system has been operational for about two years on VAX/VMS. Since a year it is also available for MS-DOS systems. CLiP (and its predecessor VAMP) has been used for a variety of middle size programs (10k - 30k lines of code). The textprocessing environments range from the formatters Runoff and Latex to the word-processors Lotus Manuscript and Word Perfect. The programming languages have mainly been Fortran, Pascal, Turbo Pascal Vision and Modula-2.

We have found that the CLiP style of programming is sufficiently intuitive to be very easy to learn and use. This is at least partly due to the fact that the CLiP programmer deals with only two languages and a couple of style concepts rather than with four independent languages (see section 1). In this respect CLiP is undoubtedly superior over other literate programming approaches.

The fact that the author/programmer is free to use his own preferred text processing environment is very convenient and makes it rather easy to accept the CLiP system. CLiP imposes virtually no limits on the way a text processing system is being used. In particular CLiP supports the use of any sort of illustrations for explanatory purposes, which is generally recognized as a important advantage [Thimbleby 1986, Reenskaug 1989, Ramsey 1991]. We found it a great help being able to explain data structures not only by words, but also in terms of diagrams.

We consider CLiP's independence of a programming language a definite advantage. Together with its ability to extract several modules from a set of files it provides a much appreciated flexibility. We not only extract the program modules, but also various additional files the system may need for proper use and maintenance. The appendices display an example.

A general problem with literate programming tools is called the preprocessor problem. The compiler and debugger will give their reports with respect to the line numbers of the extracted modules rather than the original documentation. For this reason the modules have to be examined next to the documentation and the references are indirect. This is a nuisance, especially if the layout of the modules has little correspondence to the code lines of the documentation. In our view Knuth is definitely wrong in asserting that the intermediate modules can (and in fact should) be ignored, the reason why his WEB system produces deliberately unreadable modules [Knuth 1984]. CLiP does not format the code fragments, neither in the documentation nor in the extracted modules. Therefore the code fragments look very much alike in both situations. Although this does not eliminate the preprocessor problem of course, it makes it much easier to deal with [Ramsey 1991].

The most prominent disadvantage of CLiP is its inability to produce an index of program identifiers fully automatically. This problem is inherent to CLiPs language independence. With CLiP one has to create an X-ref list the same way as one would create the index of a book. Thus the text processing system completely determines the degree of support for this activity.

If CLiP is used with a word-processor, one has to export ASCII-files from the word-processor files before CLiP can proceed. This implies a small amount of overhead which we consider neglectable.

6 Future developments

The quest for techniques to write comprehensible programs started in the early seventies with structured programming [Dijkstra 1972, Wirth 1971/1974]. Literate programming is a significant step forwards, formulated by Lins as "Literate programming = structured programming + structured documentation" [Lins 1989]. What more can we expect in the future?

A fairly obvious idea is to replace the book format by hypertext structures. Experiments indicate that the way in which information is disclosed by means of different access paths is of eminent importance for its comprehensibility [Oman 1990a/1990b]. Since a hypertext is a generalization of the conventional book format there can be no doubt that it provides a superior paradigm in this respect. In addition a hypertext easily accommodates a multimedia approach to code explanation. It would for instance be possible to explain the behaviour of a given data structure by an animation in moving images rather than by a description in plain English.

It is possible to formalize the design steps that we make to (de)compose a system in terms of a particular model. For instance Back and Morris have formalized the stepwise refinement technique as proposed by Dijkstra and Wirth in the early seventies [Wirth 1971/1974, Dijkstra 1972, Back 1980, Morris 1987]. Let us consider the programming process as a sequence of design steps each of which is implemented by a certain amount of code. In the context of a model it is possible to validate the code of every step with respect to the corresponding design decision and to support the validation process by means of a tool.

The metaphor of a literate program as a textbook can be carried through even further to that of a mathematical textbook. A mathematical textbook does not explain a theory in terms of theorems that are proven by formal techniques. Instead it derives its proofs by 'informal rigour', that is the proofs are written in natural language but in a very precise and unambiguous formulation. Using the same technique a literate program can, at least in principle, be modelled as an informal correctness argument for the implemented system. This seems an attractive thought especially in a context where (de)composition steps can be validated by automatic tools. In our department we are exploring this path.

7 Conclusions

Our experiences with literate programming in general and with CLiP in particular are very positive. CLiP is easy to use and its flexibility is very much appreciated. We fully confirm the observation of others that the improvement in quality of the final product by far outweighs the initial overhead that inevitably goes with producing a program in literate form [Knuth 1984, Levy 1993]. The advantages are even more prominent when it comes to the maintenance of a program.

All of the present literate programming tools have originated from people in need of a module extractor for at best a limited variety of environments. This is considered a serious drawback to make literate programming a generally accepted technique [Van Wyk 1990]. CLiP is unique in that it has been designed to be a truly general literate programming tool. Consequently the system misses a few features, but this is not experienced as a serious limitation.

8 References

Ammers E.W. van and M.R. Kramer, VAMP: A Tool for Programming Independent of Programming Language and Formatter, *Proceedings of the 6th Annual Computer Conference CompEuro'92, The Hague, 371-376*.

Avenarius A. and S. Oppermann (1990), FWEB: A Literate Programming System for Fortran8x, *ACM Sigplan Notices 25, 1, 52-58*.

Back R.J. (1980), Correctness Preserving Program Refinements: Proof, Theory and Applications, MC Tract 131, *Mathematical Centre Tracts*, Amsterdam.

Bentley J. and D.E. Knuth (1986a), Programming Pearls: Literate Programming, *Communications of the ACM 29, 5, 364-369*.

Bentley J., D.E. Knuth and D. McIlroy (1986b), Programming Pearls: A Literate Program, *Communications of the ACM 29, 6, 471-483*.

Bishop J.M. and K.M. Gregson (1992), Literate Programming and the LIPED Environment, *Structured Programming 13, 1, 23-34*.

Brooks F.P. (1982), The Mythical Man-Month: Essays on software Engineering, *Addison Wesley*, Reading, Massachusets.

Dijkstra E.W. (1972), Notes on Structured Programming, pages 1-82 in Structured Programming (O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare), *Academic Press*, London.

Fox J. (1990), Webless Literate Programming, *TUGboat 11, 4, 511-513*.

Knuth D.E. (1984), Literate Programming, *The Computer Journal, 27, 2, 97-111*.

Knuth D.E. (1986), TeX: The Program, *Addison Wesley*.

Levy S. (1987), WEB Adapted to C: Another Approach, *TUGboat 8, 1, 12-13*.

Lins C. (1989), A First Look at Literate Programming, *Structured Programming 10, 1, 60-62*.

Morris J.M. (1987), A Theoretical Basis for Stepwise Refinement and the Programming Calculus, *Science of Computer Programming 9, 287-306*.

Oman P.W. and C.R. Cook (1990a), The Book Paradigm for Improved Maintenance, *IEEE Software 7, 1, 39-45*.

Oman P.W. and C.R. Cook (1990b), Typographic Style is More than Cosmic, *Communications of the ACM 33, 5, 506-520*.

Parikh G. and N. Zvegintov (eds.) (1983). Tutorial on Software Maintenance, *IEEE/ Computer Society Press*, Silver Spring, Md.

Parnas D.L. and C. Clemants (1986), A Rational Design Process: How and Why to Fake It, *Software Engineering 12, 2, 251-257*.

Ramsey N. (1989), Weaving a Language Independent WEB, *Communications of the ACM 32, 9, 1051-1055*.

Ramsey N. and C. Marceau (1991), Literate Programming on a Team Project, *Software Practice and Experience 21, 7, 677-683*.

Reenskaug T. and A.L. Skaar (1989), An Environment for Literate Smalltalk Programming, *OOPSLA'89 Proceedings, 337-345*.

Sewell E.W. (1987), How to MANGLE your Software: The WEB System for Modula-2, *TUGboat 8, 2, 118-122*.

Sewell E.W. (1989), Weaving a Program: Literate Programming in WEB, *Van Nostrand Reinhold*.

Smith L.M.C. and M.H. Samadzadeh (1992), Measuring Complexity and Stability of WEB Programs, *Structured Programming 13, 1, 35-50*.

Soloway E, J. Pinto, S. Letovsky, D Littman and R. Lampert (1988), Designing documentation to Compensate for Delocalized Plans, *Communications of the ACM 31, 11, 1259-1267*.

Thimbleby H. (1986), Experiences of ′Literate Programming′ using Cweb (a variant of Knuth's WEB), *The Computer Journal 29, 3, 201-211*.

Tung S.-H. (1989), A Structured Method for Literate Programming, *Structured Programming 10, 2, 113-120*.

Van Wyk C.J. (1990), Literate Programming: An Assessment, *Communications of the ACM 33, 3, 361-365*.

Wirth N. (1971), Program Development by Stepwise Refinement, *Communications of the ACM 14, 4, 221-227*.

Wirth N. (1974), On the Composition of Well-Structured Programs, *ACM Computing Surveys 6, 4, 247-259*.

Appendix A: Palindrome filter

In this appendix we illustrate the CLiP style of literate programming by a program to filter palindromic lines from an input file. Three files are involved: the program module (`PALINDROME.PAS`), a test file (`TESTDATA.TXT`) and a command file (`PALINDROME.COM`) to run the palindrome filter.

Subsections A.1 through A.7 display the documentation literally, but we have numbered the code lines in the margin for easy reference from the main text.

A.1. Specification

A palindrome is a sentence with the property that the letters from left to right, read the same as from right to left. In the comparison uppercase and lowercase letters are considered to be equivalent and all other characters are simply ignored. Hence an empty sentence is a palindrome. Other examples are:

```
 1    (*************  #file "TESTDATA.TXT"  #comment off  *************)
 2    Ada
 3    1234567
 4    Able was I, ere I saw Elba.
 5    A man, a plan, a canal, Panama.
 6    Norma is as selfless as I Am, Ron.
 7    (*****************  End of TESTDATA.PAS  ************************)
```

The program `PALINDROME` reads an input file, filters the lines that are palindromic and writes them to an output file.

A.2. Communication with the outside world

The program conforms to the general template of a Pascal program. We introduce the files `IN_FILE` and `OUT_FILE` to define its communication with the outside world. The actual files have to be specified at run-time. Thus we have

```
 8    (*****************  #file "PALINDROME.PAS"  ********************)
 9    (***************************************************************)
10    (* Program: Palindrome filter program.                         *)
11    (* Purpose: To filter the palindromic lines from a given input  *)
12    (*          file to a specified output file.                   *)
13    (***************************************************************)
14    PROGRAM PALINDROME (INPUT, OUTPUT, IN_FILE, OUT_FILE);
15
16    (*******    Palindrome constants  #multiple #comment off  *******)
17    (*******    Palindrome types      #multiple #comment off  *******)
18    VAR
19        IN_FILE, OUT_FILE: TEXT;
20    (*******    Palindrome variables  #multiple #comment off  *******)
21
22    BEGIN
23        RESET   (IN_FILE);
24        REWRITE (OUT_FILE);
25
26        (*****************  Palindrome (body)  ********************)
27        (** Copy the lines of the IN_FILE that are palindromic to  **)
28        (** the OUT_FILE.                                          **)
29        (*********************************************************)
30
31    END (*PALINDROME*).
32    (******************  End of PALINDROME.PAS  *******************)
```

To prepare the module for future declarations of constants and types we have

```
(*******        Palindrome constants      #leader #quick  *******)
CONST
(*******        Palindrome types          #leader #quick  *******)
TYPE
```

A.3. Processing of the files

The program processes `IN_FILE` line by line. The idea is to buffer an exact copy of the current line in `IN_LINE`, while at the same time its letters are buffered in `LETTERS`. So `LETTERS` will be empty if the line holds no letters at all, in which case the line is considered to be palindromic by definition.

We choose the buffers `IN_LINE` and `LETTERS` to be of the same type, `TEXT_LINE`, which we will not specify in detail right now. For this purpose we introduce a type `ABSTRACT`.

```
(*******        Palindrome types                  #quick  *******)
    ABSTRACT  = (DEFINED, UNDEFINED);
```

`TEXT_LINE` will temporarily be declared `ABSTRACT` and its details will be defined later. Thus the declaration of `TEXT_LINE`

```
(*******                Palindrome types                  *******)
(*******            Declaration of TEXT_LINE              *******)
(*****************  End of Palindrome types  ********************)
```

is temporarily satisfied with the type `ABSTRACT`.

```
(*******    Declaration of TEXT_LINE     #quick #default  *******)
    TEXT_LINE = ABSTRACT;
```

The declaration for the variables `IN_LINE` and `LETTERS` becomes

```
(*******    Palindrome variables                  #quick  *******)
    IN_LINE,
    LETTERS:        TEXT_LINE;
```

We have to test `LETTERS` in order to decide whether or not `IN_LINE` contains a palindrome. The result of this test is flagged by `IS_PALINDROME`, for which we introduce the declaration

```
(*******    Palindrome variables                  #quick  *******)
    IS_PALINDROME:  BOOLEAN;
```

Now the body of the Palindrome filter may be expanded as

```
    (*****************  Palindrome (body)  *********************)
    (** Copy the lines of the IN_FILE that are palindromic to  **)
    (** the OUT_FILE.                                          **)
    WHILE NOT EOF (IN_FILE) DO
    BEGIN
        (*****************  Palindrome (1)  ********************)
        (** Read a line from IN_FILE into IN_LINE. The letters **)
        (** of this line are copied to LETTERS.                **)
        (*******************************************************)

        READLN (IN_FILE);

        (*****************  Palindrome (2)  ********************)
        (** Test palindromicity of LETTERS. Set IS_PALINDROME  **)
        (** to reflect the result of the test.                 **)
        (*******************************************************)

```

```
            IF IS_PALINDROME THEN
            BEGIN
                (*****************  Palindrome (3)  ****************)
                (** Write IN_LINE to OUT_FILE.                     **)
                (**************************************************)

                WRITELN (OUT_FILE);
            END (*IF*);
        END (*WHILE*);
        (*************  End of Palindrome (body)  ******************)
```

A.4. Choosing the structure of IN_LINE and LETTERS

Before we can proceed we need to establish a structure for the objects IN_LINE and LETTERS. Thus we define TEXT_LINE as a structure with two components. The first component is an array, CHARS, which contains the characters to be buffered. The second component, LENGTH, indicates which part of the array is actually occupied. The maximum number of characters that can be buffered by the structure is determined by the length, MAX_L, of the array. MAX_L serves as an implementation parameter.

```
   (*******    Palindrome constants                  #quick  *******)
       MAX_L = 132;

   (*******        Declaration of TEXT_LINE          #quick  *******)
       TEXT_LINE =     RECORD
                           CHARS: ARRAY[1..MAX_L] OF CHAR;
                           LENGTH: 0..MAX_L;
                       END (*RECORD*);
```

A.5. Reading a line

For efficiency reasons we fill IN_LINE and LETTERS simultaneously. Therefore we buffer every character that is read from IN_FILE in the variable IN_CHAR.

```
   (*******    Palindrome variables                  #quick  *******)
       IN_CHAR:    CHAR;
```

Only when IN_CHAR turns out to be a letter it is copied to LETTERS. Since this process is crucial for the overall operation, we make provisions for some debugging code here.

```
            (*****************  Palindrome (1)  ********************)
            (** Read a line from IN_FILE into IN_LINE. The letters **)
            (** of this line are copied to LETTERS.                **)
            IN_LINE.LENGTH := 0;
            LETTERS.LENGTH := 0;
            WITH IN_LINE DO
            WHILE NOT EOLN (IN_FILE) DO
            BEGIN
                READ (IN_FILE, IN_CHAR);
                LENGTH := LENGTH + 1;
                CHARS[LENGTH] := IN_CHAR;
                IF IN_CHAR IN ['A'..'Z', 'a'..'z'] THEN
                WITH LETTERS DO
                BEGIN
                    LENGTH := LENGTH + 1;
                    CHARS[LENGTH] := IN_CHAR;
                END (*WITH/IF*);
            END (*WHILE/WITH*);

            (********************  Palindrome (test)  **************)
            (** Check contents of IN_LINE and LETTERS.  #optional  **)
            (********************************************************)

            (****************  End of Palindrome (1)  **************)
```

A.6. Testing for palindromicity

We test the palindromicity of LETTERS in two steps. First we transform the contents of LETTERS to uppercase and then we compare the characters of LETTERS pairwise. The comparison is done starting with the most outside characters and progressing inward. The string is assumed a palindrome until the opposite is proven through a pair of different characters. With the local counter

```
(*******    Palindrome variables                  #quick  *******)
    I:        INTEGER;
```

we keep track of the comparing process. Now Palindrome (2) can be expanded as

```
        (*****************  Palindrome (2)  ********************)
        (** Test palindromicity of LETTERS. Set IS_PALINDROME  **)
        (** to reflect the result of the test.                 **)
        WITH LETTERS DO
        BEGIN
            (* Transform lowercase to uppercase.                *)
            FOR I := 1 TO LENGTH DO
            IF CHARS[I] IN ['a'..'z']
            THEN CHARS[I] :=
                 CHR(ORD(CHARS[I]) - ORD('a') + ORD('A'));

            (* Perform the palindromicity test.                 *)
            IS_PALINDROME := TRUE;
            I := 1;
            WHILE IS_PALINDROME AND (I <= LENGTH DIV 2) DO
            IF CHARS[I] = CHARS[LENGTH-I+1] THEN
                I := I + 1
            ELSE
                IS_PALINDROME := FALSE;
        END (*WITH*);
        (*****************  End of Palindrome (2)  **************)
```

A.7. Writing the palindrome

The only remaining action is to write the contents of IN_LINE. Again we need a local counter

```
(*******    Palindrome variables                  #quick  *******)
    J:        INTEGER;
```

The writing proceeds straight forward.

```
            (*****************  Palindrome (3)  *****************)
            (** Write IN_LINE to OUT_FILE.                     **)
            WITH IN_LINE DO
            BEGIN
                FOR J := 1 TO LENGTH DO
                    WRITE (OUT_FILE, CHARS[J]);
            END (*WITH*);
            (*************  End of Palindrome (3)  **************)
```

A.8. Running the Palindrome filter

To run the program in a VAX/VMS environment the following command procedure is convenient.

```
(*********  #file "PALINDROME.COM"  #comment off  ***************)
$!***************************************************************!
$! Run PALINDROME. Input file and output file are parameters.   *!
$! Par1: Specification of input file.                           *!
$! Par2: Specification of output file.                          *!
$!***************************************************************!
```

```
$DEFINE IN_FILE  'P1
$DEFINE OUT_FILE 'P2
RUN PALIND
(****************  End of PALINDROME.COM  *********************)
```

Appendix B: Testcode

This appendix is a separate file. It contains the test code that can be inserted to debug the palindrome filter program. If the PALINDROME.PAS module is extracted from appendices A and B together, then the resulting module includes this test code.

B.1. Print the contents of IN_LINE and LETTERS

Correct reading of the input is crucial. For debugging purposes we may want to inspect the contents of IN_LINE and LETTERS. We need a local counter

```
(*******    Palindrome variables              #quick  *******)
    T :         INTEGER;
```

We want the debugging information te be clearly flagged as such.

```
(*****************  Palindrome (test)  ******************)
(** Check contents of IN_LINE and LETTERS.             **)
WRITELN;
WRITELN ('============  DEBUGGING INFORMATION  ===============');
WRITELN ('Contents of buffer IN_LINE: ');
WITH IN_LINE DO
FOR T := 1 TO LENGTH DO WRITE (OUTPUT, CHARS[T]);
WRITELN ('Contents of buffer LETTERS: ');
WITH LETTERS DO
FOR T := 1 TO LENGTH DO WRITE (OUTPUT, CHARS[T]);
WRITELN ('==========  END OF DEBUGGING INFORMATION  ==========');
WRITELN;
(*************  End of Palindrome (test)  ***************)
```