Experience with Literate Programming or Towards Qualified Programming

Thomas Setz

Technische Hochschule Darmstadt
Fachbereich Informatik
Alexanderstr. 10
D-64283 Darmstadt
Germany

thsetz@cdc.informatik.th-darmstadt.de

Abstract This article illustrates the experience gained in the field of literate programming while developing the distributed system LiPS over a period of 6 years. A big part in this area of research is system programming, and after the first version of the system which was implemented in "pure C" I was looking for a better way to realize the system. I thought the approach of literate programming which views programming as writing literature and therefore merges programming language and documentation language in one document, would be a good choice and started to use CWEB; a tool for literate-programming with the C language.

We started with the CWEB version 3.0 [KL93] and soon found that we did not like the restriction to only use the TeX documentation language. Therefore, we integrated a possibility to use LaTeX as the documentation language. With the increased usage of WWW documents and the hypertext markup language (HTML) we added mechanisms to follow hypertext links in documents like a Master Thesis to the "real" program in the CWEB file. This enables us to follow a more abstract thought and its description in a "upper level" document to the real intrinsics of its implementation using standard tools like NETSCAPE and is the basis to look at the whole project as a big hypertext document.

While using this tool on a daily basis, we found that literate programming should be enhanced. It is not enough to write well documented programs, they should also be tested in order to reach better software quality. That is what we call qualified programming.

In this article we first give a short introduction to programming with CWEB. The next sections deal with additional tools and practical issues like GNU Emacs support and automatic translation to HTML within the LiPS development system. The status of our current work, the enhancement of literate programming towards qualified programming, is presented next. We finish this article with a view on our future work.

1 Introduction

In 1991 I started implementing a distributed system called LiPS which uses the idle-cycles in networks of workstations for the purpose of distributed computations [Set91,SR93,Set96,SF96,SL97]. A big part in this area of research is system programming, and after the first version of the system which was implemented in "pure C" I was looking for a better way to realize the system. I heard about the literate programming approach and took a look at it.

I started reading the article "Literate Programming"[Knu84] written by Donald Knuth where he says:

> I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be *works of literature*. Hence, my title: 'Literate Programming'. ... Let us change our traditional attitude to the construction of programs: 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. ... surely nobody wants to admit writing an *illiterate* program.

After having read so far in the article , I liked the idea of sitting in my office late in the night writing on a scene in the second act of my comedy (drama) called LiPS where a system process (the good one) suddenly is stroken by a fault (the bad one). I found it a nice way to look at the problem – although the more technical view, namely what happens if the `ioctl()` does not work, seemed to be closer to the real world. Anyway, what I missed having read so far was a better way of integrating a hierarchy into the code. I knew the DeMarco approach [DeM79] of Structured Analysis, where a complex problem is subdivided into smaller subproblems, so-called bubbles, and each of these subproblems

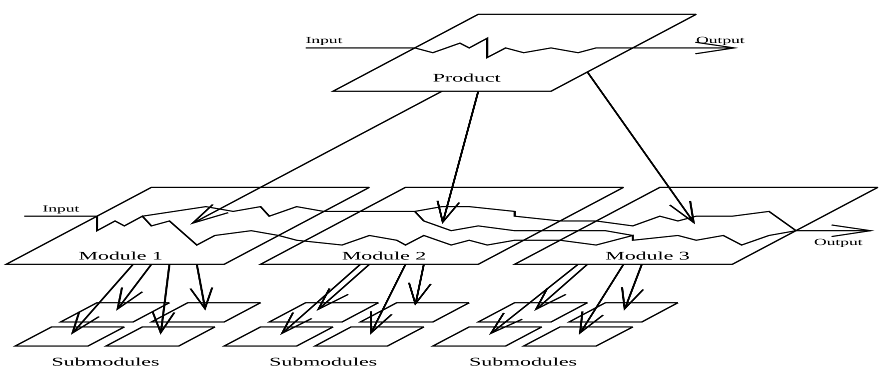


Figure1. Layering Data Flow Diagrams

could be subdivided further into additional layers until the so-called level of mini-specifications is reached, close to the level of real implementation. This scenario is depicted in Figure 1. Reading further in Knuth's article I found

> I chose the name WEB partly because it was one of the few three-letter words of English that hadn't already been applied to computers. But as time went on, I've become extremely pleased with the name, because I think that a complex piece of software is, indeed, best regarded as a *web* that has been delicately pieced together from simple materials. We understand a complicated system by understanding its simple parts, and by understanding the simple relations between those parts and their immediate neighbors. If we express a program as a web of ideas, we can emphasize its structural properties in a natural and satisfying way.

There is the "structural property" I was looking for.

Since then we started to develop the whole system (approximately 60.000 lines of documented code) in CWEB. Using the CWEB tool on an every day basis, we adapted it to our needs. The first step was the possibility to use LaTeX as documentation language instead of TeX. Then we integrated the CWEB documents nicely into the imake-, autoconf- and CVS-based development system and created template files for manual pages, header files etc. . We introduced commands to integrate additional glossary and keyword listings into the CWEB files and added a bibliography to those files. In the next step we found that it would be good to follow the hierarchy of a CWEB document via hypertext links. Especially, it should be possible to follow the hierarchy across the border of the document itself. This enables us to follow an abstract view of an approach, for example, in a Master Thesis via a hypertext link to its implementation in another HTML file being generated from the CWEB-based source code. Currently, we are working on a CWEB-based test environment helping us to (automatically) decide whether the piece of literature (module) is found to be a drama or comedy on the stage (platform).

In this article I first give an example of writing and structuring a CWEB program and show the different translation tools needed to generate program code as documentation from a CWEB file. In the next section our Emacs interface, specially adapted to our needs, is sketched. The generation of HTML code from a CWEB file is explained next, and some examples show how we use this feature. The following section sketches the state of our current work dealing with the integration of a test language into the CWEB document. Preceding the conclusion and summary, we give some performance data for the usage of the different translation tools.

2 Programming in CWEB

The CWEB package is a front-end to the C programming language; it is not an entire new system. So everyone familiar with programming in C will be able to write code in CWEB. A CWEB program holds both documentation and C code in one file, so the system is helpful to improve structured documentation. The documentation is written in LaTeX style, and therefore every CWEB file is a mixture of LaTeX and C code. First I will give a very small example of the simplest way to write a CWEB program.

```
@
@c
#include <stdio.h>
int main()
{
        printf("Hello, world!\n");
        return 0;
}
```

The only difference between the CWEB file (typically ending with `.w`) and the well known Hello world example in C can be found in the first two lines. The first `@` sign introduces the documentation part of the CWEB Hello world program and the `@c` starts the C part of the program. The minimal difference between a C file and a CWEB file is `@ @c`.

In the rest of this section I will first introduce a more complex program making use of the documenation and structuring capabilities of CWEB. Then, a high level view on the structure of a CWEB file is presented. The section closes with the description of tools needed in the process of translating a CWEB file into the different document and program representations.

2.1 Developing More Complex Programs

It is not only possible to write normal C code in CWEB programs, but it is also possible (and strongly recommended!) to build a structured program using bubbles.[1] In the example, we develop a small utility called "findstr" which outputs the location of the first occurrence of a string in a file. We will now examine how we can divide the program into small units. As shown in Figure 2, a natural approach would be to split the file into a bubble holding the necessary include files and another one holding the main program. The main part needs to perform input in order to get the name of the file, to open it and read it into memory. Then it has to do some computation in order to find the string in the memory. At the end, it prints the result of the analysis.

The code for the implementation of the main part in CWEB will be presented next. In the example, we omit the coding for the manual page and included header files and start with the main part.

[1] According to the CWEB terminology, the structuring units are called sections but I prefer the term bubble for a section as it is closer to the DeMarco terminology.

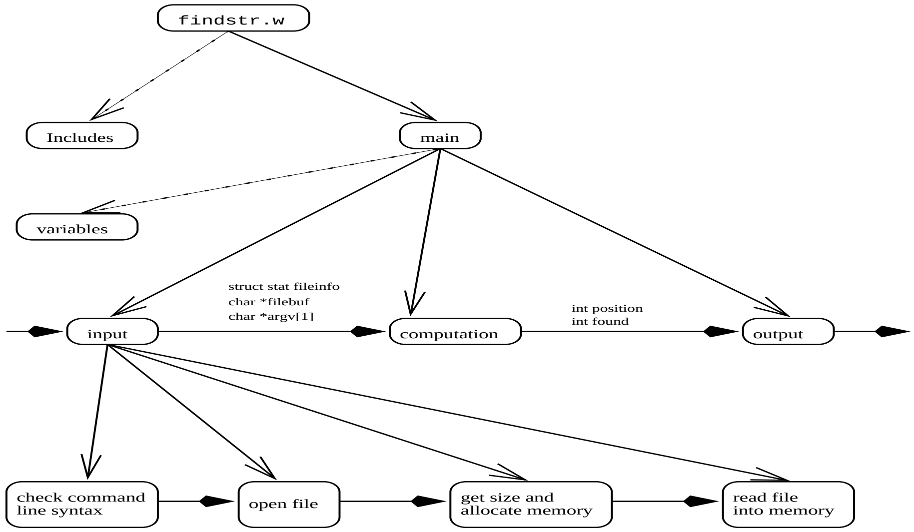


Figure2. A sample program tree

```
@ The Program findstr. \newline
We use this straightforward procedure: Examine the command
line, get the arguments and check if they are in correct
format. Then read the given file, search for the string and
output the result. If an error occurs, the program will exit
with an error message.

@c
int main(int argc, char **argv) {
        @<variable declarations@> @/
        @<input (get string and file)@> @/
        @<computation (search string occurrence)@> @/
        @<output (print result on screen)@> @/
        exit(0);
}
```

So we see that a bubble is created by the code "`@< name @>`". Now that the bubbles are declared, we can define their contents. This is done in the following way: We open a new section, write the documentation code, and then we do not use the "`@c`" sequence to start the C code part. Instead, we use the construct "`@< name @>=`". This also starts the C part of a section, but now the following C code is taken as the content of the bubble called "name". Let's take a look at how we fill the input bubble:

```
@ Input. \newline
We will get the arguments from the command line and read the
given file. It is easy to get the arguments: they can be
accessed by any C program in the function main() as argc,
which holds the number of arguments, and argv, which is an
array of strings that were given in the command line. If the
format of the command line turns out to be wrong or if we are
unable to read the given file, we output an error message and
exit.

@<input (get string and file)@>=
@<check command line syntax@> @/
@<open file@> @/
@<get size of file and allocate memory@> @/
@<read file into memory@> @/
```

So you see how the usage of bubbles work: in a bubble, you may of course define more bubbles, which you can define later. Then, at the bottom of the program structure tree, you will use real C code and that may look like this.

```
@ The |stat()| system call will get file information into the
|fileinfo| structure. The entry |st_size| in that structure
indicates the total file size in bytes.  Then we allocate as
much memory as we need to hold the complete file. If anything
fails, we exit with an error message.

@<get size of file and allocate memory@>=
if (stat(argv[2], &fileinfo)!=0) {
        fprintf(stderr, "cannot get size of file %s\n",
                argv[2]);
        exit(-1);
}

filebuf=malloc(fileinfo.st_size);
if (filebuf==NULL) {
        fprintf(stderr, "cannot allocate %d bytes\n",
                fileinfo.st_size);
        exit(-1);
}
```

The code for the other missing bubbles, is omitted as it should be easy to imagine how the rest of the application could be coded. Having understood this concept, you should now be able to write your own CWEB programs.

2.2 A High Level View on a CWEB File

A high level view on a CWEB file is given in Figure 3. It is easy to see that the file (module) consists of mutiple bubbles each of which is further divided into a documentation part and a code part. Within every code part it is possible to "call" additional bubbles, which could be defined later. The limbo part at the

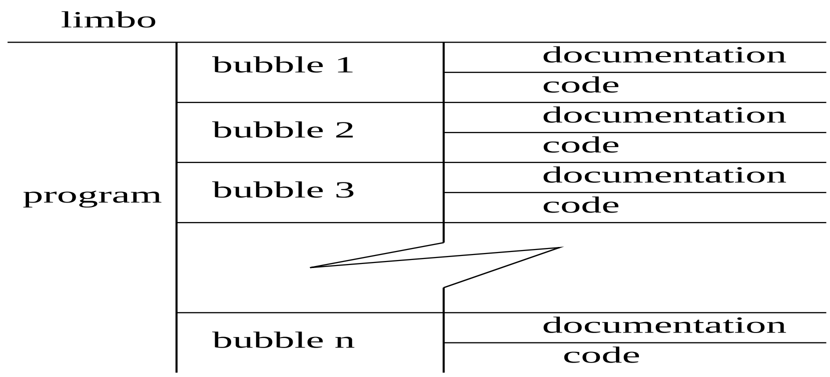


Figure3. Structure of a CWEB file

beginning of a CWEB file is an area in which one may use plain TEX commands as definitions, page settings or other things.

2.3 Translating a CWEB File Into Other Formats

As shown in Figure 4, the translators `ctangle` and `hycweave` are used to get the documentation or program code from the CWEB file. The `ctangle` command extracts the C code from the `.w`-file and throws away all comments. The resulting `.c`-file can be fed to a C compiler like cc or gcc. The documentation is prepared using the command `hycweave` which creates a `.tex`-file that can be fed to LATEX to create DVI files or to LATEX2HTML in order to build an HTML document.

2.4 Advanced Control Codes

For a complete catalogue of CWEB control codes, please refer to the document "The CWEB System of Structured Documentation" by Donald E. Knuth and Silvio Levy, which is shipped with the CWEB package. A special feature worth mentioning is the highlighting of variable names in the documentation part of a bubble. If you mention a variable, function, or section in the LATEX part of a bubble, you may want to write "`|example_function()|`" instead of the ordinary way. This will yield nicer output, and in addition the index will be searched

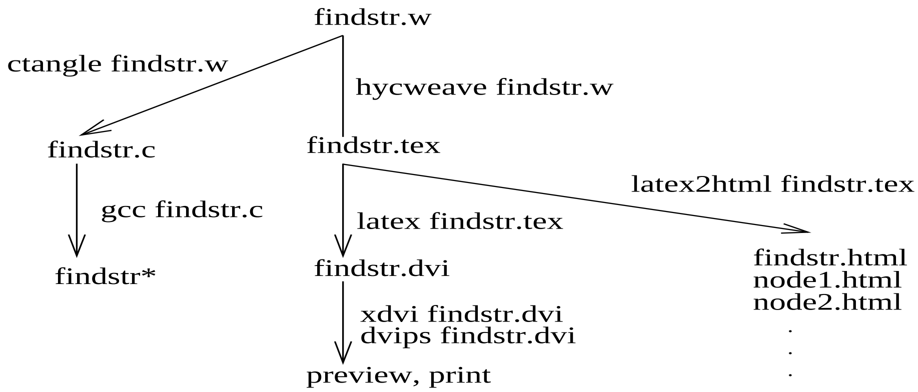


Figure4. How to work with CWEB

to print the section number where `example_function` is defined after the expression.

3 The Directory Structure of LiPS

The TOP directory of the LiPS system as shown in Figure 5 consists of several subdirectories for the different types of files in the system. The `cweb/` directory which is further divided into subdirectories holds the LiPS system code written in CWEB. When we generate the system, further subdirectories named according to the directory name in the CWEB directory are created. The `src/` subdirectory will later contain the C code of the program, and the `platform/` directory, divided into system specific subdirectories, will then contain system dependent object code, libraries and executables. The `doc/` directory holds the TeX, DVI, PostScript and HTML representation of the CWEB files. Again, the name of each subdirectory is chosen according to the directory name in the CWEB tree. Directories that are needed beside the initial directories in the CWEB tree are generated on the fly and therefore do not need to be part of the revision control system. The `inc/` directory holds the necessary header files which are also generated from CWEB files. The `Install_and_work` directory holds some directories where LaTeX sources for documentation like theses or papers live. The `figs/` directory holds subdirectories of all pictures used somewhere in the documentation. This feature enables us to reuse pictures once drawn in multiple places.

4 Emacs Modes

GNU Emacs is much more than an editor. Together with its Lisp interface and the variety of mode packages that come with the distribution, it provides the abil-

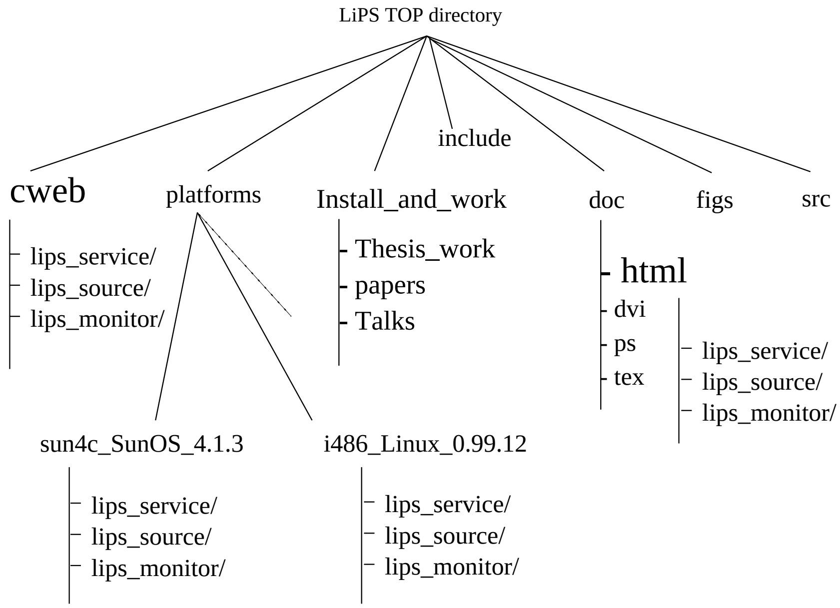


Figure5. The LiPS directory tree

ity of editing with centralized customization. Meanwhile, we consider it a mandatory component for our project development. With our LiPS Emacs package we provide a standardized interface for PERL, C, LaTeX, and CWEB documents. Figure 6 shows Emacs with the file `findstr.w` using the LiPS-CWEB-mode. By applying Lisp hooks to basic modes such as the PERL-mode we tune in standard definitions such as indentation widths, short cuts or hot keys for special LiPS editing features. But also basic faces as text highlighting with the `hl319` package, or umlaut handling with the `iso-accents-mode` are incorporated. Additionally we have a special Emacs mode for CWEB files. It's main features are:

- Mouse support for the X version of the GNU Emacs, so that you can easily jump between bubbles;
- Hypertext-like browsing through CWEB code;
- Im- and exploding of bubbles, to hide/show code/documentation part(s) of bubble(s) in the edit buffer;
- Highlighting of CWEB code;
- Built-in shell to `ctangle`, `hycweave`, LaTeX and C-compilers;
- Adaptations for the usage with the LiPS development environment, e.g. insertion of standard CWEB header files or manual pages.

This makes editing especially, large CWEB documents, very comfortable.

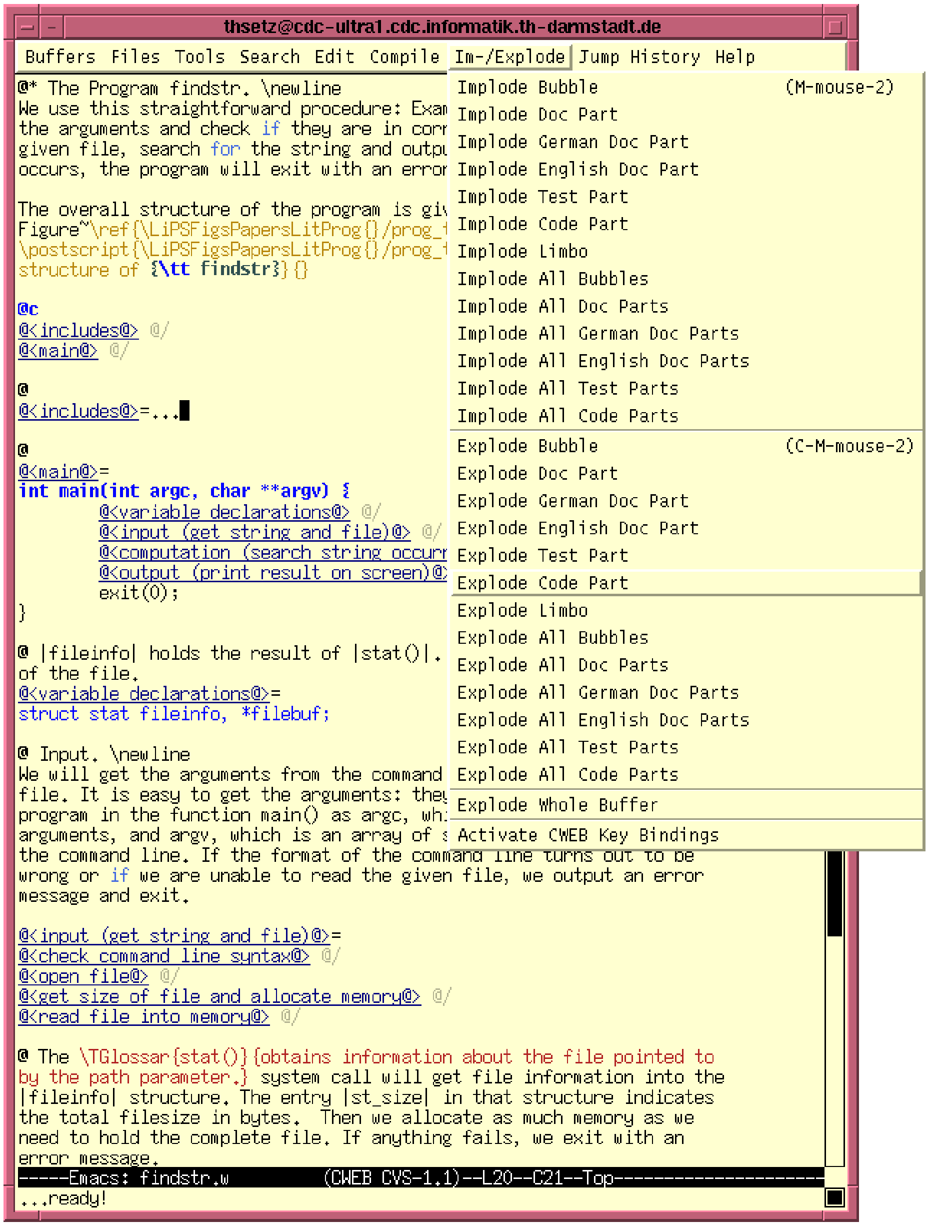


Figure6. Emacs on the screen

5 Developing for the Internet

The World Wide Web has reached wide-spread use and the up-to-date representation of a project has meanwhile become a major concern. Given LaTeX2HTML, a translator from LaTeX to the hypertext markup language (HTML) it is easy to compile the whole project to a World Wide Web representation.

5.1 LaTeX2HTML Briefly

LaTeX2HTML was initially developed by Nikos Drakos, University of Leeds (Great Britain).

LaTeX2HTML is a conversion tool based on the PERL programming language that allows documents written in LaTeX to become part of the World Wide Web. In addition, it offers an easy migration path towards authoring complex hypermedia documents using familiar word-processing concepts.

LaTeX2HTML replicates the basic structure of a LaTeX document as a set of interconnected HTML files as it is shown in Figure 7 which can be explored using automatically generated navigation panels. The cross-references, citations, footnotes, the table of contents and the lists of figures and tables are also translated into hypertext links. Formatting information which has equivalent "tags" in HTML (lists, quotes, paragraph-breaks, type-styles, etc.) is also converted appropriately. The remaining heavily formatted items such as mathematical equations, pictures or tables are converted to images which are automatically placed at the correct positions in the final HTML document. The conversions are summarized in Table 1.

It extends LaTeX by supporting arbitrary hypertext links and symbolic cross-references between evolving remote documents. It also allows the specification of conditional text and the inclusion of raw HTML commands. These hypermedia extensions to LaTeX are available as new commands and environments from within a LaTeX document.

LaTeX	HTML
text passage	text passage
math formula	GIF image or HTML MATH
tabular	GIF image or HTML TABLE
figure	GIF image

Table1. LaTeX2HTML's simplification for HTML.

Figure 8 shows how the LaTeX2HTML converter is realized. [2] It is written in the

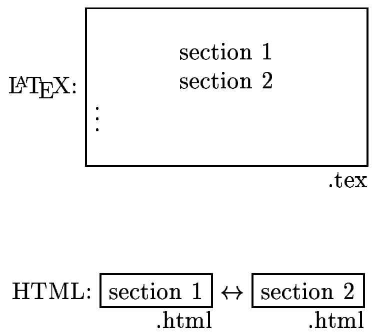


Figure7. A possible replication of the document structure.

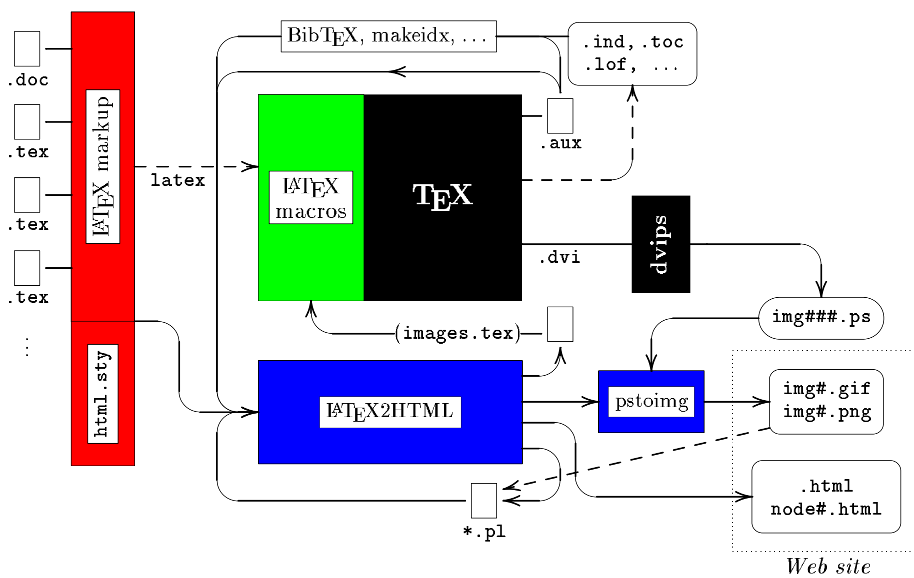


Figure8. Realization of the LaTeX2HTML converter.

PERL programming language and uses various other programs and tools such as LaTeX, `makeindex` and `pstoimg`.

5.2 Conversion of CWEB to HTML

Together with LaTeX2HTML and our CWEBTEX package, we are able to generate an HTML presentation from the `hycweave` output. This raises the hyperization of a CWEB file to its full powers.

Within the HTML document, we may jump back and forth between refinements, specific locations of variables, the index or the glossary, and much more. Everything that is hyperizable with CWEB is available in our HTML presentation. Additionally, we have now a presentation ready for the Internet.

Figure 9 and 10 show two examples of CWEB files converted to HTML – both a high level description and a (low level) CWEB bubble with typical hypertext elements.

5.3 Linkage of CWEB Documents

It would be nice to point from one refinement of a specific module to another refinement located elsewhere. With our `hycweave` alone, this would only be possible if the refinements lived within one CWEB file.

Within our project, we have an overwhelming amount of CWEB documents which are also quite heterogeneous. This rules out the approach to have a top level CWEB file which includes all the underlying ones.

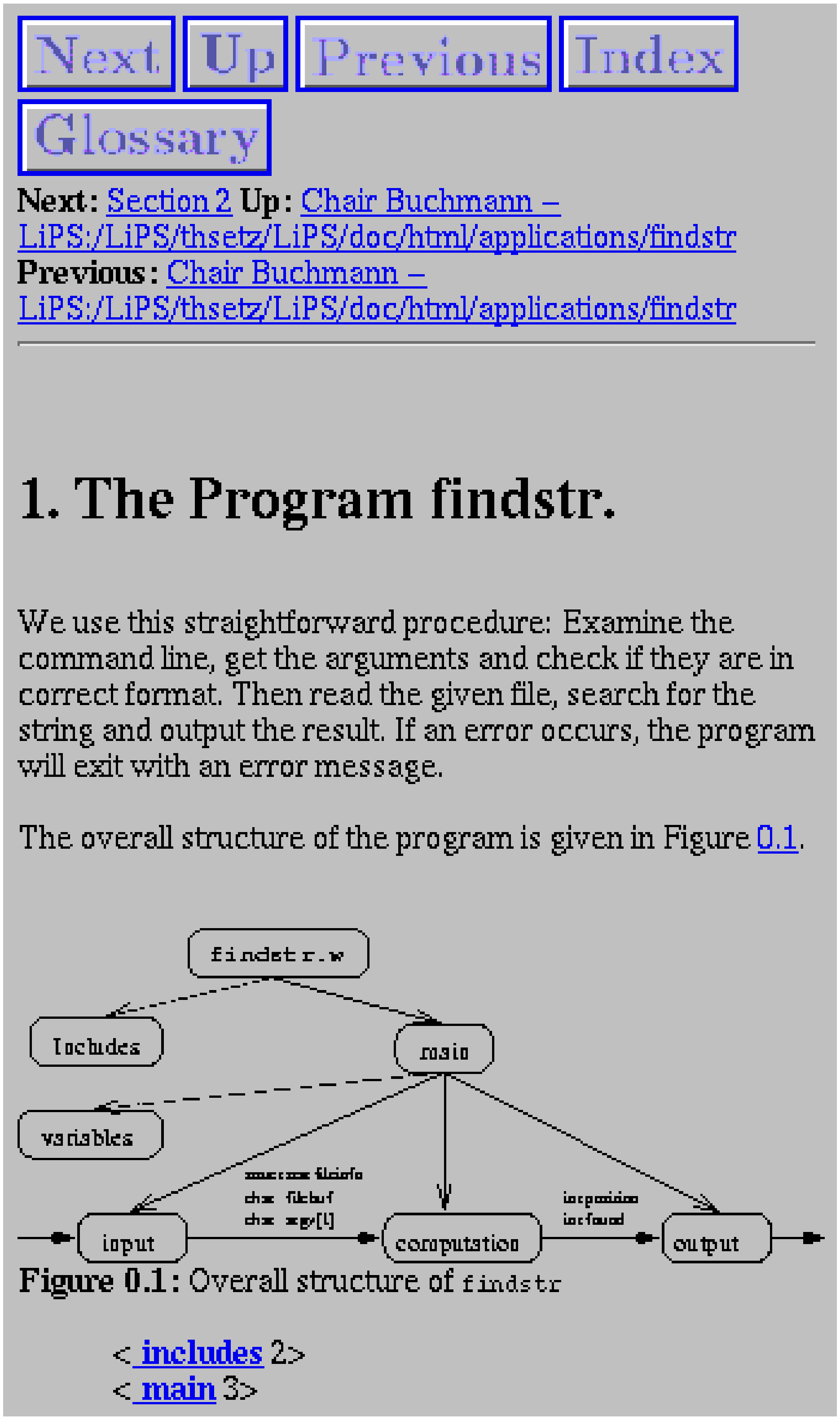
Consequently, our next step resulted in linkage of stand-alone CWEB files. We developed some special TeX macros which, provided we have unique refinement names, enables us to point to a refinement outside the current CWEB file. We call this an inter-refinement.

This feature is rendered in the DVI output as well as available with HTML to jump between CWEB presentations (Figure 11 on page 16).

6 Current Work

So far, we have built an environment well-suited to implement and document our system. But implementing and documenting is not the only thing to be done while building a distributed system. Code changes over the time, bugs are fixed and code is ported to other architectures. Needless to say that some code being originally implemented on one architecture will not work on another one or even worse, may not behave as expected. The same condition holds for bug fixes,

[2] The figure is taken with kind permission of Ross Moore (ross@mpce.mq.edu.au) from his talk at the first LaTeX2HTML workshop at the Technische Hochschule Darmstadt, Germany.



Next Up Previous Index Glossary

Next: Section 2 **Up:** Chair Buchmann – LiPS:/LiPS/thsetz/LiPS/doc/html/applications/findstr
Previous: Chair Buchmann – LiPS:/LiPS/thsetz/LiPS/doc/html/applications/findstr

1. The Program findstr.

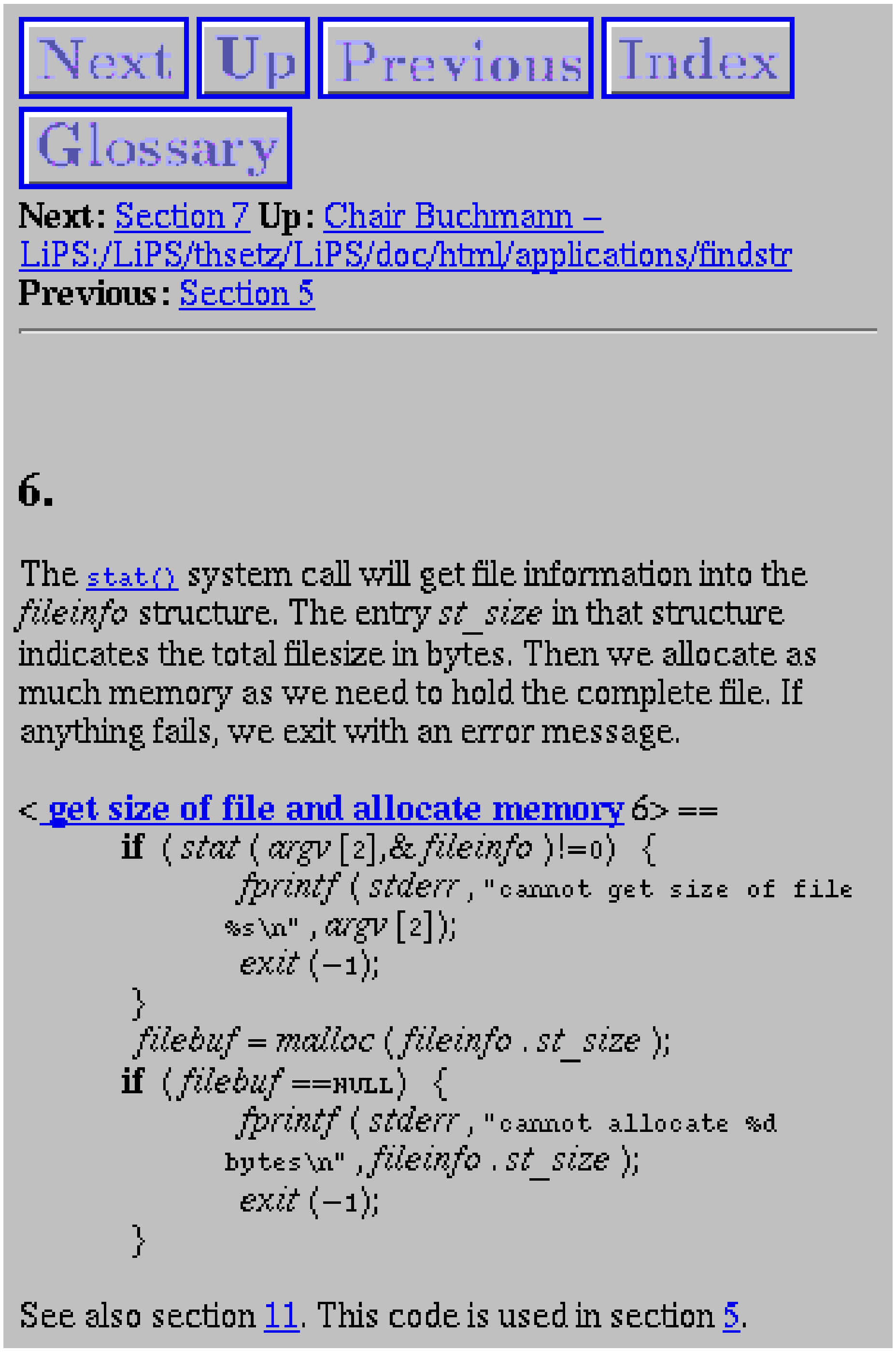
We use this straightforward procedure: Examine the command line, get the arguments and check if they are in correct format. Then read the given file, search for the string and output the result. If an error occurs, the program will exit with an error message.

The overall structure of the program is given in Figure 0.1.

Figure 0.1: Overall structure of `findstr`

< **includes** 2>
< **main** 3>

Figure9. NETSCAPE showing a high level description of the `findstr` program



Next Up Previous Index Glossary

Next: Section 7 **Up:** Chair Buchmann – LiPS:/LiPS/thsetz/LiPS/doc/html/applications/findstr
Previous: Section 5

6.

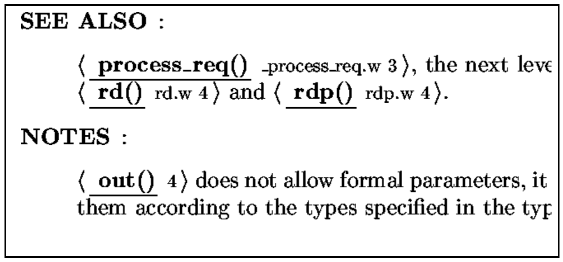
The `stat()` system call will get file information into the *fileinfo* structure. The entry *st_size* in that structure indicates the total filesize in bytes. Then we allocate as much memory as we need to hold the complete file. If anything fails, we exit with an error message.

< **get size of file and allocate memory** 6> ==

```
if (stat(argv[2],&fileinfo)!=0) {
        fprintf(stderr,"cannot get size of file %s\n",argv[2]);
        exit(-1);
}
filebuf = malloc(fileinfo.st_size);
if (filebuf==NULL) {
        fprintf(stderr,"cannot allocate %d bytes\n",fileinfo.st_size);
        exit(-1);
}
```

See also section 11. This code is used in section 5.

Figure10. HTML representation of a (low level) CWEB bubble



SEE ALSO :

⟨ **process_req()** _process_req.w 3 ⟩, the next leve
⟨ **rd()** rd.w 4 ⟩ and ⟨ **rdp()** rdp.w 4 ⟩.

NOTES :

⟨ **out()** 4 ⟩ does not allow formal parameters, it
them according to the types specified in the typ

Figure11. Refinement links of file `out.w`
You see links to inter-refinements (e.g. to refinement `rd()` *of file* `rd.w`*) and a link to an intra-refinement (*`out()` *of the current file).*

which will make the system work on one architecture but may introduce some more errors on another architecture. It is hard work to find out what the error is and why it appears. This field asked for more tools to be integrated into the development environment.

In the first step, we wrote a tool called `spectest` [STea94], which is able to generate a test program from a C function and a test description (Figure 12). The test description is written in a test description language, defining the test

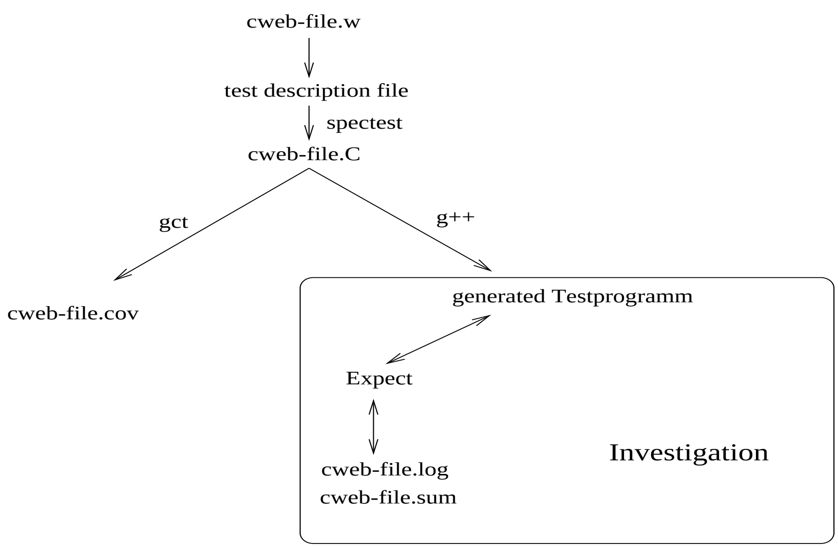


Figure12. How to work with the test environment

case by its preconditions, the tests to be performed and the expected results to be reached. Multiple tests can be performed with one test description. It is possible to define stubs for functions being called from the tested functions in order to investigate the tested function in isolation (unit test). It is also possible to call the "real" function and thereby making an integration test. The main advantage of our `spectest` tool in comparison to other test tools in this area, e.g. dejagnu [Sav96], is the possibility to perform the tests on the basis of a function instead of being able only to test a main program.

In the next step we integrated this tool together with gct [Mar95], Expect [Lib94], g++ [Sta94] and our development environment [STea94]. This is still ongoing work, and a more detailed description is given in [Lip97]. The integration of gct enables us to find the coverage of the performed tests. The coverage measurements determine whether the set of tests applied to the module have test cases such that every branch of a function is walked through at least once while the tests are performed. The integration of our development environment and Expect enable us to simply type `make findstr.test`, and all tests for this module are performed automatically. After the tests are finished, a file named `findstr.sum` gives a summary on the performed tests. Its content looks like:

```
-- Test Summary for /LiPS/lippmann/LiPS/test/examples/findstr.w
------------------------------------------------------------
-- Working revision:     Repository revision:
-- RCS Id: findstr.w,v 1.1 1997/01/22 11:28:19 lippmann Exp


UNITTESTfindstr.t: PASS
test1: PASS
test2: PASS

BINARY BRANCH INSTRUMENTATION (4 conditions total)
1 (25.00%) not satisfied.
3 (75.00%) fully satisfied.

LOOP INSTRUMENTATION (3 conditions total)
2 (66.67%) not satisfied.
1 (33.33%) fully satisfied.

MULTIPLE CONDITION INSTRUMENTATION (4 conditions total)
2 (50.00%) not satisfied.
2 (50.00%) fully satisfied.

SUMMARY OF ALL CONDITION TYPES (20 total)
9 (45.00%) not satisfied.
11 (55.00%) fully satisfied.
```

A more detailed description, especially a description of what failed if the test failed, are given in `findstr.log`. The coverage of the tests are given in `findstr.cov`.

7 Performance

As I found in some discussions, a lot of people tend to think that the additional documentation in the programming document takes too much time while developing programs for translation between CWEB and C, I built up a table with the timings spent within the different tools. This list is given in Table 2. It shows the times (sum of user and system CPU time) of the different tools on various platforms for translating a file in the order of magnitude of `findstr`. It can be seen that the times additionally needed by the CWEB tools (ctangle and hycweave) can be ignored in comparison to the others.

	ctangle	gcc	hycweave	LaTeX	LaTeX2HTML	dvips
sun sparc ultra 170E	0.0	0.3	0.0	1.2	1:03.9	0.1
sun sparcstation 4	0.0	1.5	0.3	4.2	58.7	0.5
sun SLC	0.1	6.1	0.1	17.1	4:53.7	6.7

Table2. Time measurements for the different tools on different platforms

8 Conclusion and Summary

In this article, I have presented how literate programming is used within the development of our system LiPS. I showed the similarity of structuring properties in established software engineering methods like Structured Analysis to structuring possibilites given in programming with CWEB, and how this is integrated into our project's development system. The property to translate CWEB documents and accompanying documents like Master Theses into HTML format, and thereby enabling us to see the whole procect as a large hypertext document, shows the analogy.

The integration of software testing into our development system should contribute to a better quality of our software, although many questions of how testing should be realized come up and have to be solved in the future.

I have been working for a couple of years with the literate programming aproach, and keep going on. In [Knu84] Donal Knuth said about his opinion on literate programming

> In fact, my enthusiasm is so great that I must warn the reader to discount much of what I shall say as the ravings of a fanatic who thinks he has just seen a great light.

I have already joined this party a couple of years ago, and with the hypertext and testing extension – the direction towards qualified programming – the light even seems to be brighter now.

9 Acknowledgements

Building a distributed system and a nicely fitting development environment is a lot of work. While working at the Universität des Saarlandes, Saarbrücken (Germany), I had a lot of students involved in the implementation of our development system. I am very grateful to them spending lots of hours in the integration and adaption of tools into the development system.

Martin Tews wrote the first version of `hycweave`[3] and integrated the LaTeX documentation language. Thomas Liefke wrote the first version of the `spectest` tool which now is the backbone of our test environment. Harald Lorchert realized the first version of our Emacs mode. Jens Lippmann spent a lot of time in implementing the translation of CWEB files to HTML and integrated the test environment into our development system.

References

[DeM79] T. DeMarco. *Structured Analysis and System Specification.* Prentice-Hall publishers-Yourdon, Inc, 1979. Also published in/as: Yourdon, Inc., New York, 1978.

[KL93] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0.* Addison-Wesley, Reading, MA, USA, 1993.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[Lib94] Don Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs.* O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, December 1994.

[Lip97] Lippmann J. *Integration einer Testumgebung in* LiPS. Diplomarbeit, Universität des Saarlandes, Lehrstuhl Prof. Buchmann, 1997.

[Mar95] Brian Marick. *The Craft of Software Testing.* Prentice Hall, 1995.

[Sav96] Rob Savoye. *The DejaGNU Testing Framework.* Free Software Foundation, http://www.cygnus.com/library/dejagnu/dejagnu_toc.html edition, 1 1996.

[Set91] Setz T. *Integration einer Linda-orientierten Laufzeitumgebung in* LiPS. Diplomarbeit, Universität des Saarlandes, 1991. Fachbereich Informatik, Lehrstuhl Professor Buchmann.

[Set96] Setz T. *Integration von Mechanismen zur Unterstützung der Fehlertoleranz in* LiPS. PhD Thesis, Universität des Saarlandes, 2 1996. Fachbereich Informatik, Lehrstuhl Prof. Buchmann.

[SF96] Setz T. and Fischer J. Software Fehlertoleranz vom Level Eins in LiPS. In *Proceedings of SIWORK'96, Workstations and their applications*, Zürich, May 1996.

[3] This version changes the original `cweave` only by some 10's of lines.

[SL97] Setz T. and Liefke T. The LIPS Runtime Systems based on Fault-Tolerant Tuple Space Machines. In *Proceedings of the Workshop on Runtime Systems for Parallel Programming (RTSPP), 11th International Parallel Processing Symposium (IPPS'97), Geneva, Switzerland*, April 1997. Appeared as Technical Report, Vrije Universiteit Amsterdam, Faculteit der Wiskunde en Informatica, No. IR-417, februari 1997.

[SR93] Setz T. and Roth R. Distributed Processing with LIPS. In *ALCOM*, Saarbrücken, August 1993.

[Sta94] Stallman R. M. *Using and Porting gcc.* Free Software Foundation, 1994.

[STea94] Setz T., Tews M., and et al. *The* LIPS *Development System*, 10 1994. Universität des Saarlandes, Fachbereich Informatik, Lehrstuhl Prof. Buchmann.