Incremental redocumentation using literate programming

Nwe Nwe Win

Incremental Redocumentation Using Literate Programming

Nwe Nwe Win

M.Sc. Thesis

Centre for Software Maintenance
Department of Computer Science
University of Durham

1998

Abstract

The primary aim of this research is to investigate means of improving program comprehension through redocumentation. In particular it will concentrate on using Literate Programming as a method for program redocumentation.

Documentation is crucially important as an aid to understanding software systems. The Incremental Redocumentation Using Literate Programming System analyses the existing source code and merges in a range of other information, in order to create a complete documentation package. This may include not only traditional paper documents, but also hypertext facilities, animated specifications and output from other analysis tools. The status of the documentation is implicitly elevated to that of an integral part of the system, rather than an optional extra. Where a configuration management system is used to manage different versions of a system, the documentation can also be brought under its control.

The literate programming paradigm provides the encouragement and capability to produce high quality code and documentation simultaneously. Conceptually, literate programming systems are document preparation systems. The primary goal of a literate program is to be understandable to the programmers who are going to have to read it at some later date – often while involved in maintenance, or perhaps when trying to determine the possibility of reusing parts of the code for later projects.

This thesis presents a structures of C programs and literate C programs, and describes the features of captured literate C programs. A method of the capture process and also the functions of the redocumentation process are described. In addition, this thesis outlines how the individual stages in the capture process and the edit process are used to redocument a C program. The results of application of the process are highlighted by way of example programs. The evaluation process is performed by comparing the results of an existing literate program with those resulting from the application of the method described within this thesis. The results have shown that the captured redocumented literate C program is more readable and understandable than source code only, and that it provides a basis for subsequent maintenance and further redocumentation.

Acknowledgements

The author would like to acknowledge the Ministry of Education, Myanmar, and the British Council for the award of research studentship.

Special thanks are also due to my supervisor Mr. Malcolm Munro, Department of Computer Science, University of Durham, for all his help and guidance throughout the course of this research, and Dr. Kyaw Thein, Rector, and all the colleagues at the Institute of Computer Science and Technology, Myanmar, for their support and encouragement.

At last, not the least, the author would like to thank Professor Keith Bennett and all the colleagues at the Centre for Software Maintenance, University of Durham, United Kingdom for all their support, useful comments and facilities provided.

Statement of Copyright

Contents

Chapter 1

Introduction

Program comprehension is an important aspect of software maintenance being relevant across the range of maintenance activities [42]. A large percentage of software engineering activity is in dealing with existing software [33]. Software maintenance, reuse, reverse engineering, and re-engineering are expensive activities. A significant part of this cost is in program comprehension. There are many activities necessary to make the comprehension process possible and meaningful. Documentation is crucially important as an aid to understanding a system.

If developers can document their reasoning for design decisions, how various parts of the system are intended to interact, and their thoughts about future program modification, then this too can greatly reduce the problems of program comprehension once a system enters the maintenance phase. Even when code is designed so that changes can be carried out efficiently, the design principles and design decisions are often not recorded in a form that is useful to future maintainers. Documentation is the aspect of software engineering most neglected by both academic researchers and practitioners. Their interests are short-term interests, and their work satisfaction comes from developing programs.

When documentation is available, it is usually poorly organised, incomplete and imprecise [16]. Often a programmer or manager decides that a particular idea is clever and write a memo about it while other topics, equally important, are ignored. In other situations, where documentation is a contractual requirement, a technical writer, who does not understand the system, is hired to write the documentation. The resulting documentation is ignored by the maintenance programmers because it is not accurate. Some projects keep two sets of books: there are the official documentation, written as required for the contract, and the real documentation [37].

Documentation that seems clear and adequate to its authors is often unclear to the programmer who must maintain the code later. Even when the information is present, the maintenance programmer does not know where to look for it. It is almost as common to find that the same topic is covered twice, but that the statements in the documentation are inconsistent with each other and the code.

Documentation is not an *attractive* research topic. Software documentation has always been

given low priority status compared to other software activities [18]. In talking to people developing commercial software we find that documentation is neglected because it will not speed up the next release. Again, programmers and managers are so driven by the most imminent deadline, that they cannot plan for the software old age. If we recognise that software aging is inevitable and expensive, that the first or next release of the program is not the end of its development, that the long-term costs are going to exceed the short term profit, we will start taking documentation more seriously.

A major step slowing the aging of older software, and often rejuvenating it, is to upgrade the quality of the documentation. Often documentation is neglected by the maintenance programmers because of their haste to correct problems reported by customers or to introduce features demanded by the market. When they do document their work, it is often by means of a memo that is not integrated into the previously existing documentation, but simply added to it. If the software is really valuable, the resulting unstructured documentation can, and should, be replaced by carefully structured documentation that has been reviewed to be complete and correct [37].

In general, documentation is anything which provides information about the software system. Thus it includes the source code, Job Control Language (JCL), test suites, as well as development documents, designs, user documentations and code analysis results. The contents of documents may include natural language text, diagrams, mathematical expressions, and maybe in the form of flowing text, tables, structured diagrams etc.[49] However understanding is achieved, comprehension of the code should be considered as only part of the process. There must also be a mechanism whereby the information recovered is retained for use. If maintainers can incrementally document their understanding of a system, in such a way that the time taken to understand that area of code for future modifications is vastly reduced, the net result will be realisation of significant savings in maintenance costs.

The Incremental Redocumentation Using Literate Programming System [47] analyses the existing source code and merges in a range of other information, in order to create a complete documentation package. This may include not only traditional paper documents, but also hypertext facilities, animated specifications and output from other analysis tools. The status of the documentation is implicitly elevated to that of an integral part of the system, rather than an optional extra. Where a configuration management system is used to manage different versions of a system, the documentation can also be brought under its control.

The literate programming paradigm provides the encouragement and capability to produce high quality code and documentation simultaneously. Conceptually, literate programming systems are document preparation systems. The primary goal of a literate program is to be understandable to the programmers who are going to have to read it at some later date – often while involved in maintenance, or perhaps when trying to determine the possibility of reusing parts of the code for later projects [27].

Hypertext [12] has the potential of being a useful basis for the development of a tool for the redocumentation of existing software systems. The power of cross-referencing between related components of documentation and between differing levels of documentation has already been recognised as valuable in hard-copy software documentation. Hypertext as a technology offers the capabilities of integrating these ideas into an interactive environment.

Analysis tools can be used to gain a detailed understanding of a program. The understanding may be gained using a variety of methods. The knowledge gained is very often not recorded for the benefit of future maintenance. Hence, the gained knowledge will be added to the documentation incrementally. In the interactive system hypertext will be used to support cross-referencing, consistency and readability of a program.

1.1 Research Method

The primary aim of this research is to investigate means of improving program comprehension through redocumentation. In particular it will concentrate on using Literate Programming as a method for program redocumentation.

Conceptually, literate programming systems are document preparation systems. The literate programming paradigm provides the encouragement and capability to produce high quality code and documentation simultaneously. The goal is to create literate programs which have some facilities such as table of contents, cross-reference and indexes to help readers to comprehend the programs quickly and thoroughly. Literate programming is concerned with chunks of code.

The research method to be used will first explore the Literate Programming paradigm and identify how this paradigm can be applied to existing programs. Programs written in the programming language C will be redocumented. The language C was chosen as there are a large number currently being written in C and these programs are in general undocumented. Analysis of C programs will have to be carried out to identify the chunks of literate programming. In a C program, chunks can be for example a set of includes, defines, the global variables or the functions.

Evaluation of the success of this research will be through a comparison of redocumented C programs with those C programs written using the literate programming paradigm.

A C program is captured to a literate C program by the capture process. The captured literate C program has cross-reference for chunks and identifiers and also printed type set documentation by using LaTeX. But this is only a basic achievement of the system. The stages of the incremental redocumentation process are as follows.

1.2 The Criteria for Success

The work presented in this thesis will explore the redocumentation process and in particular how it relates to Literate Programming. The criteria for success, to be judged in the final chapter, are as follows.

1. the provision of a description of existing documentation methods and an evaluation of their suitability;

2. an exploration of existing literate programming techniques;
3. the development of a method for redocumenting C programs to produce literate C programs;
4. a description of the redocumentation (edit) process.
5. the development of prototype tools to create literate C programs;
6. an evaluation of the results of the application of the method and tool developed based on comparisons with existing literate programs;

1.3 Overview of the thesis

The remainder of this thesis is structured in the following manner.

Chapter 2 introduces the types of documentation, literate programming and discusses why redocumentation is needed. Hypertext and other literate programming systems using hypertext are described.

Chapter 3 presents the structures of C programs and literate C programs, and the features of captured literate C programs. The method of the capture process and also the functions of the redocumentation process are described. The results of application of the process to example programs are presented.

Chapter 4 outlines how the individual stages in the capture process and the edit process are used to redocument a C program.

Chapter 5 provides the evaluation of the system. It compares and contrasts of the literate C program, the captured literate C program and the redocumented captured literate C program.

Finally Chapter 6 summarises the system of Incremental Redocumentation using Literate Programming.

A summary is given at the end of each chapter.

Chapter 2

Documentation for Software Maintenance

2.1 Introduction

The understanding of existing software systems and poor documentation are related problems during software maintenance [16]. Good quality documentation can greatly aid the process of the understanding task for software maintainers. The maintainer approaching an unfamiliar system is typically confronted by documentation which is out of date, inconsistent, difficult to understand, and also sometimes inaccurate [13]. Therefore the maintainers mostly rely on the source code.

This chapter describes the types of documentation, explores literate programming, and gives a detailed description of the literate programming tool *noweb*. Why redocumentation is needed and the interactive environment for literate programming using hypertext are also discussed.

2.2 Documentation

Software documentation is written for a number of different kinds of reader [20]. Software engineers involved in the development and maintenance of a system require precise and detailed description of its application, but there is a difference between the documentation required by those involved in the development of the system and those involved in its maintenance. The former are concerned with the system as it is and will be, whereas the latter are looking at the system with a view to exploiting new arrangements that may not have been anticipated by the developers. Furthermore, the needs of project leaders are different from those of the software engineers and the documentation must allow for interpretations suitable for them. Support engineers, application builders, users, etc., all require different interpretations.

Software documentation has a significant effect on program understanding. Without accurate documentation the only reliable source of information about a program is the source code itself[3]. But it is difficult for the programmer to abstract the high level functionality of a complex program from the low level details given by the source code.

Understanding the functions and behaviour of a system from the code is a vital part of the maintenance programmer's task. If all programs to be maintained were well documented and clearly structured, the task of the maintainer would be much easier. The problem for most maintainers is that they have to maintain ill-documented code with no comprehensive structure[3]. The main problem in doing maintenance is that the maintainers cannot do maintenance on a system which was not designed for maintenance.

Robson, Bennett, Cornelius and Munro [42] discussed the approaches to program comprehension. There are two basic approaches to program comprehension of relatively small programs. First there is the systematic approach, where the maintainer examines the entire program and works out the interactions between various modules. This is completed before any attempt is made to modify the program. The other approach is the as-needed strategy where the maintainer learns enough about the program to commence the modifications. With the small program used in experiments, it is clear that the systematic approach is superior, but for large industrial scale programs this approach is not feasible and an adapted as-needed strategy must to be employed.

2.2.1 Types of Documentation

Younger [49] describes the Type of Documentation in 'The REDO Compendium', Reverse Engineering for Software Maintenance.

Documentation can be described in three types:

- Development documentation
- User documentation
- Technical documentation

Development documentation

Development documentation [51] is made for special purposes, for instance

- to record design decisions,
- to review a part of the development process,
- to freeze the state of a part of the development process,
- to verify the completeness and consistency of the design or

- to validate the design.

This type of documentation is produced during a software development process. Development documentation, which is reliable and of good quality, is valuable to the maintainer. Given reliable documentation, its utility is determined by traceability [8]. It is important that the documentation includes cross-references which allows the maintainer to trace through from the specification documents to the relevant design information and from here to the effected sections of code.

User documentation

User documentation [49] includes all documentation delivered to the user as part of the system package. It comprises information needed to install and use the system, ideally without invention from its developers. It is almost certain that some form of user documentation will exist, though its contents and scope may depend on the relationship between the developer and the user.

The minimal form of user documentation contains the information required to operate the system. This should include:

- pre- and post-conditions for execution,
- descriptions of input and output data,
- descriptions of user interaction procedures,
- internal file formats,
- details of any execution options and how these are selected, and
- error conditions, error messages and recovery procedures.

These may be in the form of examples, informal descriptions, formal functional specifications or combinations of these.

When an application is developed internally by an organisation for in-house use this may be sufficient documentation for the end user; installation and support will probably be carried out by the developers. However systems delivered to client organisations must be accompanied by more extensive documentation which ought to include:

- Installation instructions, unless installation is carried out by the supplier. This will give instructions about how to load the system from the distribution media, install the component files on the user disk store, and build the application from these files if appropriate.
- An overview of the system, the problem it is intended to solve, and its constraints. This may be interpreted as a requirements specification.

- The hardware and software operating environment. This will specify the computer system on which the application will run, its requirements for main memory and disk storage, also any peripheral devices required and configuration instructions when alternatives are possible.

User documentation is a potentially rich source of information about the application. It may provide information about the requirements specification for the system. It may also describe the input/output behaviour, and will almost certainly give detailed information about user interaction and/or data file formats. However, the information could well be in a descriptive form which is insufficient to provide a complete specification of any part of the system. Even so it is useful to the maintainer. At a high level, installation instructions or script files will provide an inventory of the files which makeup the application and perhaps also the relationship between them. At a lower level, the user interaction and input/output format descriptions aid the maintainer in identifying the roles of various variables in the programs. Although functional and other requirements may be described, user documentation is unlikely to give any information about how these are refined at design and implementation stages. However, the fact that these are stated will give clues about the functional constructs that a maintainer might look for when studying the system.

Technical documentation

Technical documentation [49] is generated from analysis of the application code by automatic tools. Technical documentation is useful in maintaining at the code level, and in reverse engineering.

Technical documentation tools typically generate low-level documentation such as identifier cross-reference listings, call graphs, control-flow diagrams, and also more abstract information such as data flow diagrams and structure charts. The former is useful to maintainers trying to understand the details of the code, the later in reverse engineering: in understanding the application at a more abstract level.

2.2.2 Documentation Tools

Documentation tools which require some level of user interaction can be useful not only to generate documentation, but also because the software engineer has to think about the system when he uses the tool. In that way the software engineer gradually gains more understanding. Documentation especially suited to software maintenance can be generated of documentation by reverse engineering. This can be added to the existing documentation. The amount of documentation and the number of ways to present it become very large. Therefore, it is necessary to have ways to select information and to choose the presentation format.

The objective of a documentation system for maintenance should be to support the process of system understanding and redocumentation. Many authors have suggested that source

code should include its own design documentation, to minimise the danger of inconsistencies appearing during maintenance. Knuth's method of 'Literate Programming' [27], goes further, organising an underlying file in such a way that compilable code, design document etc. can all be generated directly from it. Unfortunately, this rather complicated file has to be created using a normal text editor, as there are no tools to support its preparation. This approach of associating code with documentation has much to offer; most of the problems are resolved if the concept is reinterpreted using a database and hypertext links to separate out the various different kinds of materials.

2.3 Literate Programming

Literate programming changes the programmer's perspective from programming for a machine to explaining to other people (peers, students or examiners!) what the programmer intends the machine to do. It encompasses the idea that programming should be directed towards the human being who reads and writes programs rather than to the computer which executes them. The primary goal of a literate program is to be understandable to the programmers who are going to have to read it at some later date - often while involved in maintenance, or perhaps when trying to determine the possibility of reusing parts of the code for later projects. This section describes the concept of Literate Programming, the WEB system and the noweb system.

Literate programming is a method that encourages the production of a program whose primary purpose is to explain to a human what it does, as well as to instruct a computer what to do. Each program element is clearly explained, and is presented in an order that is best for human understanding. The writer has the freedom to introduce parts of the program as they are needed - which is not necessarily the order required for compilation.

Knuth [27] introduced the idea of literate programming with the statement :

> *I believe the time is ripe for significantly better documentation of programs and that we can best achieve this by considering programs as works of literature. Hence my title "Literate Programming". The practitioner of literate programming strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that nicely reinforce each other.*

During the late 1980s, an attempt was made to promote literate programming with a special section devoted to it in the Communications of the ACM [15]. Four literate programs were published [22, 24, 21, 34]. Although the Literate Programming section produced some debate, and each of the programs was moderated and commented upon by others, the section was closed down by van Wyk [47] in 1990.

Although literate programming is non-controversial and easily acceptable as a new idea, it has not been adopted into the main stream of design methods. One of the reasons for this

may be that it relies on special programming tools and Knuth's original literate programming system WEB.

2.3.1 WEB

Knuth introduced literate programming in the form of WEB, a tool for writing literate Pascal programs. WEB allows authors to write source code and descriptive text in a single document. It also gives authors the freedom to arrange the parts of a program in an order that helps explain how the program functions, which may not necessarily be the order required by the compiler [27]. Although it was later extended to C [31, 39], the original system was tied to Pascal and relied on batch mode processing.

WEB itself is chiefly a combination of two other languages:

1. a document formatting language and
2. a programming language.

WEB uses TEX as the document formatting language and Pascal as the programming language. A WEB user writes a program that serves as the source language for two different system routines. One way of processing is called weaving the WEB; it produces a document that describes the program clearly and that facilitates program maintenance. The other way of processing is called tangling the WEB. It produces a machine-executable program. The program and its documentation are both generated from the same source, so they are consistent with each other.

A full description of WEB83 appears in a Stanford report [26], which also contains the complete WEB program for WEAVE and TANGLE. In the WEB system there is no need to choose once and for all between top-down and bottom-up design methods because a program is best thought of as a web instead of a tree. A hierarchical structure is present, but the most important thing about a program is its structural relationships. With WEB a complex piece of software consists of simple parts and simple relations between those parts. The programmer's task is to state those parts and these relationships, in whatever order is best for human comprehension. It is not in some rigidly determined order like top-down or bottom-up [41].

WEB's complexities made it difficult to explore the idea of literate programming because too much effort was required to master the tool [41]. To compound the difficulty, different programming languages were served by different versions of WEB, each with its own idiosyncrasies.

The classic WEB expands three kinds of macros, pretty printed code for typeset output, evaluates some constant expressions, facilitates string support into Pascal, and implements a simple form of version control. Versions for languages other than Pascal offer slightly different functions and different sets of control sequences [41].

WEB uses its TANGLE tool to produce source code and its WEAVE tool to produce documentation. WEB's original TANGLE removed white space and folded lines to fill each line with tokens, making its output unreadable [27]. Later adaptations preserved line breaks but removed other white space. WEB's WEAVE divided a program into numbered "sections", and its index and cross-reference information referred to section numbers, not page numbers. WEB worked poorly with LaTeX. LaTeX constructs cannot be used in WEB source, and getting WEAVE output to work in LaTeX documents required tedious adjustments by hand. WEAVE's source (written in WEB) is several thousand lines long, and the formatting code was not isolated.

2.3.2 Other Literate Programming Tools

After Knuth's WEB literate programming was introduced, Thimbleby [45, 46] implemented a Unix version of it, called CWEB. CWEB is a tool to facilitate high-quality program documentation in a combination of C (the programming language) and troff (a text-formatting language).

In 1986 a project began to investigate whether literate programming could be made more accessible through a new tool by Bishop and Gregson [7]. During 1987 and 1988, a new, production-quality version of the system was implemented and distributed, and at the same time the method of literate programming was investigated with several different languages, namely Pascal, C [46, 31], Modula-2 [43] Ada [38], LISP [38], Fortran [1], Clipper and Assembly language [7]. Hyman [23] discussed the application of literate programming to object-oriented C++. An `awk` preprocessor was developed to store documentation along with source code. The information can be extracted just before compilation.

2.3.3 Evaluation of Literate Programming and Related WEB Tools

Instead of being structured for the ease of the compiler, a literate program is designed for ease of reading by a human. It is (ideally) organised in patterns which improve its readability to human eyes. The patterns should fit the cognitive patterns which the programmer uses when writing a new program, but more importantly, when trying to read a program written by someone-else in the distant past.

In principle, literate programming is not language dependent. It is a system for annotating and decomposing formulae of various kinds so that the formulae are recoverable. Means are also provided to format the combined annotations and formulae to a high standard of presentation and to provide derived cross-reference information.

This idea has a wide range of applications:

- for commentaries on classical literature;
- for multilingual commentaries on computer programs;

- for formal commentary on programs;
- for informal commentary on programs;
- for annotating a formal record of an interactive session.

According to Brown [10], Literate Programming provides important advantages over traditional programming in three different ways:

1. Literate Programming encourages organisation of code based on psychological rather than syntactic divisions. The code can be modularised by separating a group of statements which the programmer considers as a single logical unit into a named module. This allows for conceptual abstraction of the code, presented in an order and grouping, chosen to reduce psychological complexity. The division into WEB modules and the presentation of the modules is independent of any syntactic considerations of the high level programming language (HLL).
2. Literate Programming makes the program structure more easily visible. The conceptual abstraction mentioned above enables clarity in presenting the structure, no matter how complex, in a single module. If a procedure has three major parts , each part can be abstracted as a named module, and the procedure composed of those modules.
3. Literate Programming encourages an explanatory style of writing, which leads to more careful consideration of the details of the program. In discussing this explanatory style of writing, Knuth claims that WEB encourages the discipline of explaining and hence clarifying one's thoughts about a module as the code is written. The practice has been shown to be useful in reducing programming errors.

Although a pioneering system, WEB suffers from the following drawbacks:

1. A WEB program is a mixture of WEB commands, TEX commands and source code. The extra WEB and TEX commands in the source make the literate program less readable than it should be and tend to get in the way of development.
2. Section numbering and cross-reference information only appears in the final document and not in the source program. This means that during the development of a program the author must either have an up to date copy of the documentation, which could be expensive to produce or must work with the WEB source file itself, which is not nearly as readable as the final document.
3. WEB is designed for use with literate programs as Pascal as the base language. The modification of WEB for other languages [39, 12] reflects a complicated approach involving several stages. More modern paradigms including modular programming are not catered for.
4. WEB does not produce readable source code. Readable source code is necessary during program development, although the programmer should be only allowed to read the code and not modify it. Readable code is useful as a reference point for compiler error messages and debugging.

2.3.4 More General tool support for Literate Programming

Given these drawbacks an obvious line of attack is to improve WEB, in particular to place it in an interactive environment. Brown and Childs [10] presented a detailed proposal for such a system, a Literate Programming Environment (LPE). LPE concerned the development of an environment aimed at reducing the complexity of programming in WEB by creating a user interface allowing the programmer to interact more intuitively with the WEB program. A prototype language-independent interactive literate programming editor, LIPED, was developed by Bishop and Gregson in 1986 [7]. LIPED has been used for the development of a few medium sized real-world programs in a commercial environment, and for suites of educational software.

Bently addressed the concept of Literate Programming in three Programming Pearls columns in Communications of the ACM [4, 5, 6]. These columns demonstrated the applicability of Literate Programming to the explication and publication of programs, but ignored the implications for Literate Programming to real-life programming problems. He identified the activities of design, analysis and maintenance as the three aspects of a written program which were not well served by the traditional linear source listing, the purpose of which was to instruct the computer.

Several literate programs have been published, including

- A Small Work of Literature [28, 19]
- Printing Common Words [22]
- Processing Transactions [24]
- Expanding Generalised Regular expression [21]
- A File Difference Program [34]

However, none of these literate programs were available in machine readable form. The emphasis of the publications seemed to be on evaluating the cleverness and correctness of the program and its embedded documentation, to the exclusion of providing an executable entity that could be subjected to experimentation, use or maintenance.

2.3.5 noweb

The proliferation of literate programming tools made it hard for literate programming to enter the main stream, but it led to a better understanding of what such tools should do. Ramsey [39] introduced a literate programming tool *noweb*, to fill this niche. Freely available on the Internet since 1989, *noweb* striped literate programming to its essentials. Programs were composed of named chunks of code, written in any order, with documentation interleaved.

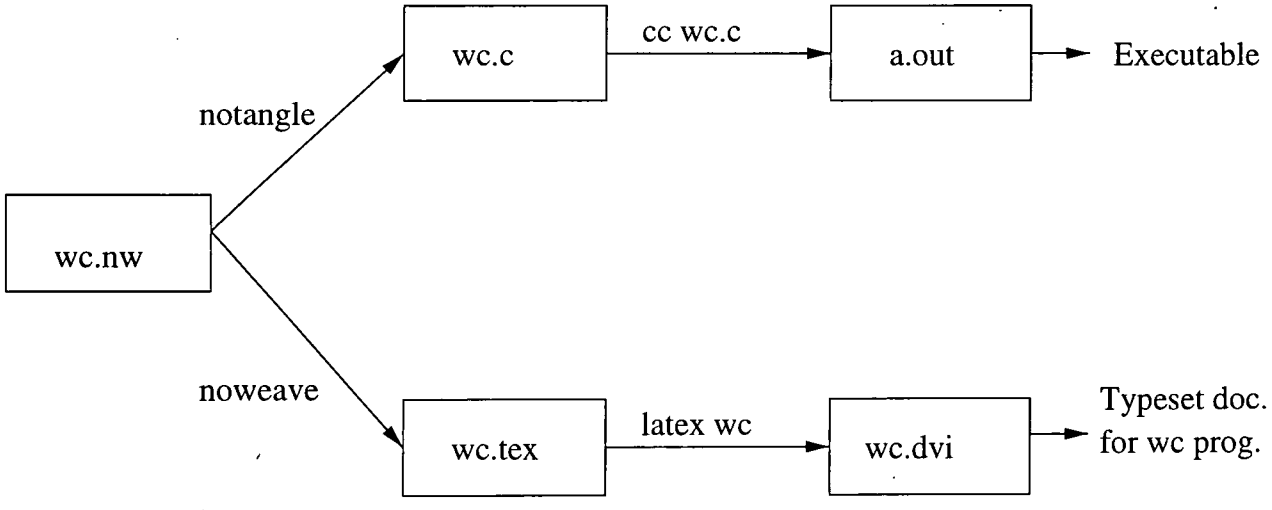

Figure 2.1: Using noweb to build code and documentation.

noweb was developed on Unix and can be ported to non-Unix platforms provided they could simulate pipelines and support both ANSI C and either awk or Icon. For example, Wittenberg [41] ported *noweb* to MS-DOS. *noweb* was unique among literate-programming tools in its pipelined, extensible implementation, which made it easy for experimenters to create new features without writing their own tools.

File structure of noweb

A *noweb* file is a sequence of chunks. A chunk may contain code, in which case it is named, or documentation, in which case it is unnamed. Chunks can be sequenced in any order. Each code chunk begins with `<<chunk names>>=` on a line by itself. The double-left angle bracket must be in the first two columns. Each documentation chunk begins with a line that starts with an `@` symbol followed by a space or new line. Chunks are terminated implicitly by the beginning of another chunk or by the end of the file. If the first line in the file does not mark the beginning of chunk, *noweb* assumes it is the first line of a documentation chunk.

As Figure 2.1 shows, *noweb* uses its *notangle* and *noweave* tools to extract code and documentation, respectively. When *notangle* is given a *noweb* file, it writes the program on standard output. When *noweave* is given a *noweb* file, it reads it and produces, on standard output, LaTeX source for typeset documentation. The *noweb* file, wc.nw, referred to in this figure is listed in Appendix A and will be used as an example in the discussion that follows.

Code chunks of noweb and their processing by notangle

Code chunks contain program source code and references to other code chunks. Several code chunks may have the same name; *notangle* concatenate their definitions to produce a single chunk. Code-chunk definitions are like macro definitions: *notangle* extracts a program by expanding one chunk (by default the chunk named `<<*>>`). The definition of that chunk contains references to other chunks, which are themselves expanded, and so on. *notangle's*

output is readable; it preserves white space and maintains the indentation of expanded chunks with respect to the chunks in which they appear. When double-left and -right angle brackets are not paired, they are treated as literals. Users can force any such brackets, even paired brackets, to be treated as literal by using preceding @ sign. Any line beginning with @ and a space terminates a code chunk. If such a line has the form @ preceding chunk defines the identifiers listed in identifiers. This notation provides a way of marking definitions manually when no automatic marking is available.

Documentation chunks and their processing

Documentation chunks contain text that is ignored by *notangle* and copied verbatim to standard output by *noweave* (except for quoted code). Code may be quoted within documentation chunks by placing double square brackets around it. These brackets are ignored by *notangle* but are used by *noweave* to give the quoted code special typographic treatment. *noweave* can work with Latex or it can use a plain TEX macro package supplied with *noweb*. *noweave* can also work with HTML, the hypertext markup language for Mosaic and the World-Wide Web. *noweave* adds no newline characters to its output, making it easy to find the sources of TEX or LATEX errors.

Index and cross-reference features

Cross-referencing of chunks and identifiers makes large program easier to understand. *noweb* does not introduce numbered "sections" for cross-reference. *noweb* uses page numbers. If one or more chunks appear on a page, they are distinguished by appending a letter to a page number. *noweb* writes chunk-cross-reference information in a footnote font below each code chunk. *noweb* also includes cross-reference information for identifiers. *noweb* generates this by using the @ source code, or by recognising definitions automatically. *noweb* uses a language- independent heuristic to find identifiers uses. It can be fooled into finding false "uses" in comments or string literals.

Compiler and debugger support

On a large project, it is essential that the compiler and other tools refer to locations in the *noweb* source, even though they work with *notangle's* output. Giving *notangle* the -L option makes it emit pragmas that inform compilers of the placement of lines in the *noweb* source. It also preserves the column in which tokens appear, so that line-and-column error messages are accurate. If the -L option is not used, *notangle* respects the indentation of its input, making its output easy to read.

How noweb supports formatting features

noweave depends on text formatters in two ways: in the source of *noweave* itself, and in the supporting macros. *noweave* dependence on its formatter is small and isolated, instead of being distributed throughout a large implementation. *noweb* uses 250 lines of source for TEX and LATEX combined, and another 250 for HTML. It uses about 200 lines of supporting macros for plain TEXand another 200 lines to support LATEX primarily because the page-based cross-reference mechanism is complex. LATEX support without cross-referencing requires only 34 lines of source and no supporting macros. HTML requires no supporting macros.

Uncoupling files and programs

The mapping between *noweb* files and programs is many-to-many, the mapping between files and documents is many-to-one. The source files can be combined by listing their names on *notangle's* or *noweave's* command line. *notangle* can extract more than one program from a single source file by using the -R command-line option to identify the root chunks of the different programs. The simplest example of one-to-many program mapping is that of putting a C header and program in a single *noweb* file. The header comes from the root chunk `<header>`, and the program from the default root chunk, `<*>`. The following Unix commands extract files wc.h and wc.c from *noweb* file wc.nw.

```
notangle -L wc.nw > wc.c

notangle -Rheader wc.nw | cpif -ne wc.h
```

noweb's `cpif -ne wc.h` compares its input to the contents of file wc.h; if they differ, the input replaces with wc.h. This trick avoids touching the file wc.h when its contents have not changed, and it avoids triggering unnecessary recompilations.

Because it is language-independent, *noweb* can combine different programming languages in a single literate program. This ability makes it possible to explain all of a project's source in a single document, including not just ordinary code but also things like make files, test scripts, and test inputs. Using literate programming to describe tests as well as source code provides a lasting, written explanation of the thinking needed to create the tests, and it does so with little overheads. If not documented at the time, the rationale behind complex tests can easily be lost.

Implementing noweb

noweb's implementation is also worth discussing, because *noweb's* extensible implementation makes it unique among literate-programming tools. *noweb* tools are implemented as pipe lines. Each pipeline begins with the *noweb* source file. Successive stages of the pipeline

implement simple transformations of the source, until the desired result emerges from the end of the pipeline.

Users change or extend *noweb* not by recompiling but by inserting or removing pipeline stages; for example, *noweave* switches from LaTeX to HTML by changing just the last pipeline stage. *noweb's* extensibility enables its users to create new literate-programming features without having to write their own tools.

noweb syntax is easy to read, write and edit, but it is not easily manipulated by programs. Markup, which is the first stage in every pipeline, converts *noweb* source to a representation easily manipulated by common Unix tools like sed and awk, greatly simplifying the construction of later pipeline stages. Middle stages add information to the representation. *notangle's* final stage converts to code, *noweave* final stage converts to TeX , LaTeX or HTML.

In the pipeline representation, every line begins with `@` and a keyword. Markup brackets chunks by `@begin ... @end`, and it uses the *noweb* source to identify text and newlines, definitions and uses of chunks, and quoted code,which can all appear inside chunks. It also preserves information about file names and defined identifiers. Other index and cross-reference information is inserted automatically by later pipeline stages.

Extending noweb

noweb lets users insert stages into the *notangle* and *noweave* pipelines, so that they can change a tool's existing behaviour or add new features without recompiling. Even language-dependent features like formatted output and automatic index generation have been added to *noweb* without recompiling.

Stages inserted in the middle of the pipeline both read and write *noweb's* pipeline representation: they are called filters, by analogy with Unix filters, which are used in the Unix implementation.

Literate programming will be used as a tool for program redocumentation.

2.4 Redocumentation

Large software systems require a different approach to software documentation than has traditionally been used. In understanding large, evolving software systems, structural redocumentation through reverse engineering plays a key role. More than 50 percent of software evolution work is devoted to program understanding [44]. The documentation task is also important. Yet documentation needs differ significantly for software systems of different scales. Most software documentation is in-the-small, because it typically describes the program at the algorithm and data-structure level. For large systems, understanding the structural aspects of the system's architecture is more important than understanding any single algorithm component [2].

Software engineers and technical managers responsible for maintaining such systems find program understanding especially problematic [48]. The documentation that exist for these systems usually describe isolated parts but not the overall structure. Moreover, the documentation is often scattered throughout the system and on different media. The maintenance personnel must explore the low-level source code and piece together disparate information to form high-level structural models. Manually creating just one such architectural document is always difficult. Creating the necessary documents to describe the architecture from multiple view points is often impossible. Yet this is exactly the sort of in-the-large documentation needed to expose the structure of large software systems.

Most documentation is written during the development of the system. Therefore, it is not sufficiently tailored to the needs of the software maintainer. Documentation is often no longer consistent with the actual state of the program. The source code itself is often the only reliable documentation [3].

Major problems faced when maintaining old software are described by Fletton and Munro in [16]. They found the following:

- Much of existing software is unstructured and is written in languages that do not easily support structured programming techniques.
- Generally maintenance programmers have not been involved in a product's development prior to maintaining it.
- The software documentation is often nonexistent, incomplete or out-of-date.

There are often a number of problems with the software documentation that is supplied to maintenance teams from the development phase of a software system. Redocumentation is the process in which new documentation is generated for an existing system either to replace or augment any documentation which already exists. The process of documentation will record the outputs from any reverse engineering that is performed. A detailed knowledge may be gained using a variety of methods, for instance analysis of code using software tools. The knowledge gained is very often not recorded for the benefit of future maintainers, resulting in repetition of effort. Failure to record the understanding gained represents a waste of resources that must be addressed. If such a documentation system is to achieve acceptance amongst experienced software maintainers, it must have a number of features as described by Fletton and Munro [16].

Incremental Documentation

Incremental redocumentation [17] is the recording of knowledge about the application as it is acquired. An essential requirement is that the documentation can be built up incrementally over a period of time without the need to worry about whether other parts of the system are documented.

Casual Update

It must be easy to update the knowledge base casually as a programmer examines source code.

Quality Assurance

It is common practice within the software industry to perform quality assurance checks on changes made to the source code. Likewise, quality assurance should be performed on any new documentation before it is incorporated permanently into the documentation base.

Team Use

Large programs often have many programmers working concurrently on maintaining the program. Therefore the documentation tool must support the use and update of the documentation base by a team of programmers.

Configuration Management

Configuration Management must be supported to allow the documentation appropriate to a particular version of the system to be recovered.

Integrated Source Code

The tool should allow the browsing of program source files in parallel with the browsing and updating of the documentation knowledge base.

Integrated Automatic Documentation

The accessibility of automatic documentation can be improved by merging the reports generated by static analysis of the source code into the documentation base.

Information Hiding

It must support information hiding to allow the documentation to be read at various levels of abstraction from the implementation that it describes.

2.5 Hypertext as a means of literate programming support

Hypertext is a way of organising information, with particularly convenient means to relate different pieces of information to each other [14]. The information to be handled by the hypertext system is divided into nodes. Each node typically stores a page of information, though most hypertext systems do not place any restriction on the size of its nodes. A document can be considered as a set of nodes with links between those nodes to form a graph. Each node contains graphical or textual information. The relationships between different pieces of information are represented using links, which tie together two (or more) nodes. The links are often anchored to specific points or regions within a node: this makes it possible to relate not only nodes, but also individual words or sentences to each other. Both nodes and links are typed to allow for different semantic interpretations of both node contents and link-relations.

The typical way of accessing information in hypertext is through navigation. At the interface level, each node appears in a window of its own, and anchors for the links relating the node to others are clearly visible. By clicking a mouse (or by some other gesture) at an anchor, the other nodes of the link relation are brought up, each in their own window: this is known as following the link. If the link is not anchored to the internal of a node, there is no visible presentation of the link. Such a link must be followed by selecting it from a menu or by similar means. Hypertext systems commonly allow attachment of attributes to both nodes and links. Such attributes are simply name-value bindings, and are often used to contain author names, keywords, last update date, etc.

The actual details of how a hypertext document is browsed and the form of the links is dependent on the actual implementation. There is an enormous variety in the hypertext systems that have been developed to date. This is not surprising given the power and generality of the hypertext concept. Bottaci and Steward [9] presented the usefulness of active links by considering how they can be used to support consistency checking during the process of writing or modifying software and its documentation. Although documenters and novelists may share a similar objective in writing for more than one kind of reader, the structure of the texts they produce are radically different. Documenters are not obliged to overload a single document with the various required interpretations, which in any case would be difficult. Typically, different documents are produced for different kinds of reader. Documentation is structured, therefore, as a number of parallel and consistent documents.

One particular approach to maintaining consistency which is applicable in some special cases is shared or included text. In a number of situations, there is a needed for several documents to contain a common item of text. If a single copy of this common item is shared amongst the several documents then any changes to the common item are automatically reflected in the item as it appears in the various documents. The mechanism of shared text is easily implemented in a hypertext and works well providing all the including documents require the same identical copy of the common item.

The structure of documentation must reflect the fact that the majority of readers will not read it in its entirety or in a single prescribed order, instead the reader is expected to select

and order relevant portions of the documentation. The term "browsing" [14] is used to describe the special style of reading appropriate to documentation. Efficient browsing is possible only if the documentation contains adequate references.

Conklin has published an extensive survey of hypertext systems [14]. A hypertext system for browsing and documenting software which make extensive use of references between code and documentation is described by Fletton and Munro [16]. Hypertext has the potential of being a useful basis for the development of a tool for the redocumentation of existing software systems. The power of cross-referencing between related components of documentation and between differing levels of documentation has already been recognised as valuable in hard-copy software documentation. Hypertext as a technology offers the capabilities of integrating these ideas into an interactive environment.

In [12] Brown describes a hypertext system to manage WEB source code. The main objective is to provide index facilities. The WEB source code is divided into a number of nodes corresponding to WEB paragraphs. A special node editor that is aware of the internal structure of WEB modules is provided. To support the different types of indexing, five different links types are supported, corresponding to different kinds of indexes.

This is a quite straight forward translation of WEB into hypertext. However, it will be beneficial to be less stringent than WEB, allowing a single code fragment to have more than one commentary, or allowing a commentary to comment on more than one code fragment.

Osterbye [36] describes hypertext programming as a way of doing literate programming that differs from using the WEB system. There is a prototype system based on the Hyperbase and Smalltalk kernel used in HyperPro [35], with a simple Smalltalk based user interface. The documentation and code are represented as hypertext. Ramsey's [41] *noweb* can work with HTML, the hypertext markup language for Mosaic and the World Wide Web (WWW). The challenge for literate programming today is getting it into use.

2.6 Summary

The objective of a documentation system for maintenance should be to support the process of system understanding and redocumentation. Types of documentation which are useful sources of information for maintainers have been identified. The concepts and structure of literate programming are described. Hypertext can be used as a tool for interactive literate programs. Redocumentation is needed for program comprehension.

Chapter 3

Incremental Redocumentation using Literate Programming

3.1 Introduction

This chapter discusses incremental redocumentation and features of *noweb* and C. The syntax of a literate C program is discussed in section 3.3.3. The capture process and the edit process are discussed in sections 3.4 and 3.5.

3.2 Incremental Redocumentation

Many people involved in maintenance feel uncomfortable because of the workload and the fact that maintenance becomes more difficult the longer the system exists. The structure of programs makes them difficult to understand and to modify. To improve the maintenance of a system is to provide methods and tools to support the comprehension of the system. Often maintenance is difficult because it is not clear where a modification of the code has to be made [50]. The objective of a documentation system for maintenance should be to support the process of system understanding and redocumentation.

Documentation is crucially important as an aid to understanding the system. The documentation which accompanies typical systems in use today is very often of poor quality and not oriented towards the needs of maintainers who frequently do not use it at all. Thus redocumentation is needed.

The process of redocumentation generates new documentation for an existing system, either to replace or supplement any documentation which already exists. The process of redocumentation will record the outputs from any reverse engineering which is performed – it is concerned with making available to the maintainer the information required to understand the system.

The literate programming concept is valuable, both for development and maintenance. It can be readily supported in a more interactive environment, to take advantage of the additional facilities which are available. Rather than weaving code and documentation in a single file, it is possible to store the two separately but to take advantage of those linking and windowing facilities which are available to maintain the intimate relationship between them. Literate programming can be used with a hypertext tool. Users familiar with the literate program prefer to work with this tool rather than the typeset output from the literate programming. In maintenance, the practice of 'literate maintenance', of documenting changes visibly and clearly at the time they are made could go some way towards reducing the degradation over time of an evolving system.

In general, the application source code will remain unchanged during the reverse engineering process. However, there are occasions when an editing facility is required, for example to manually restructure code to facilitate its future understanding or to correct coding errors. Moreover, it is also required for incremental redocumentation. The maintainers can add information on the system to the literate program gradually whenever knowledge is gained, for example by using some analysis tools.

3.3 noweb and C

The literate programming tool *noweb* was discussed in the previous chapter. A literate program consists of the combined code and documentation which can be processed and typeset to result in a high-quality presentation including a table of contents, index, cross-referencing information and typographic treatment. A literate C program is a C program written in *noweb* structure. The *noweb* structure, C layout and literate C program are discussed in this section.

3.3.1 noweb structure

Here is an overview of a literate program that is written in the *noweb* language.

```
<<*>>=
<< chunk name 1 >>
<< chunk name 2 >>
<< chunk name 3 >>
<< chunk name 4 >>
<< chunk name 5 >>
.
.
@
```

These are the code-chunk definitions of a literate program. Code-chunk definitions are like macro definitions. `<<*>>` chunk is a root chunk, and *notangle* extracts the program by expanding this chunk. The following code-chunks are also extracted by *notangle* in sequence as described in the definition. The definition of these code-chunks can be written anywhere in the program, i.e. code-chunk may appear in any order. Each code-chunk begins with `<< chunk name >>=` on a line by itself. Documentation-chunks are between `@` and beginning symbol of code-chunk `<<`. Documentation-chunks contain text that is copied verbatim to standard output by *noweave*. The expansion of the above definition demonstrates how it is possible to present chunks in any order.

```
<< chunk name 3 >>=
source code
@
documentation chunk
<< chunk name 5 >>=
source code
@
documentation chunk
<< chunk name 1 >>=
source code
@
documentation chunk
<< chunk name 4 >>=
source code
@
documentation chunk
<< chunk name 2 >>=
source code
@
```

Code chunks contain not only program source code but also references to other code chunks. References to other code-chunks may be defined within a code-chunk.

```
<< chunk name 4 >>=
source code
{
<< chunk name 4.1 >>
<< chunk name 4.2 >>
```

}

@

<< **chunk name 4.1** >>=

source code

@

documentation chunk

<< **chunk name 4.2** >>=

source code

<< **chunk name 4.2.1** >>

@

3.3.2 C layout

A C program is composed of one or more C source files [25]. Each source file contains some part of the entire C program, typically a number of external functions. Source files often have associated with them *header files* that provide declarations for the external functions used in other files. One source file must contain an external function named main; by convention this will be the program's entry point. Each source file is independently processed by a C compiler. The preprocessor is controlled by special preprocessor command lines, which are lines of the source file beginning with the character #. One of the C preprocessor commands, *#include*, inserts text from another file. . The include file may be **.h** header file and **.c** C source code file. Header files include functions, definitions, variables and type definitions.

In summary, a C program may have the following structure:

- no order
- more than one file
- separate compilation
- include may be a header(functions, definitions, variables, type definitions) and / or C code

A literate C program is different from a C program. Generally, the system intends to be language independent. The main idea will be suitable for any language, but here we use a specific programming language C as an example. One of the characteristics of a literate C program is that different parts of the program such as informal top-level descriptions like macros which are usually abbreviated.

In summary, a literate C program may have the following structure:

- Header Files

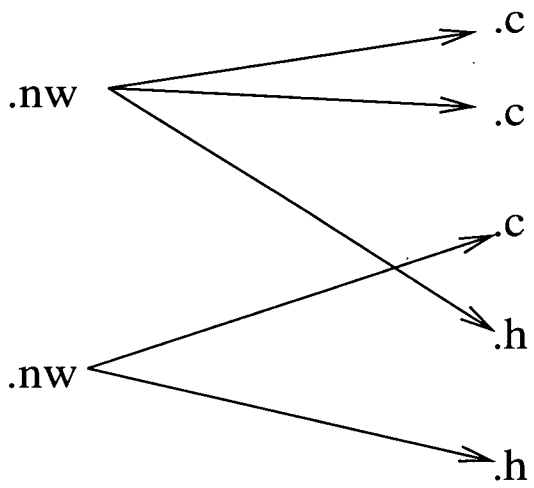

Figure 3.1: many to many

- Definitions
- Global Variables
- Functions
- Main Program

By using a *notangle* command, those top-level descriptions are replaced by their expanded meanings, and a C program will be obtained. A limitation of the literate program is that the C code is in one source code file and no separate compilation is possible. In a literate C program header file, there are no other source code other than header.

The features of a literate C program are as follows:

- order (assumed)
- all C code in one file
- no separate compilation
- include files .h (header)

The mapping between *noweb* files and programs is many-to-many; the mapping between files and documents is many-to-one. An example of many-to-many program mapping is that of putting a C header and program in a single *noweb* file. The header comes from the root chunk << *header* >>, and the program from the default root chunk << * >>. An example of many-to-one is to create a file listing all the identifiers defined anywhere in **a.nw**, **b.nw**, or **c.nw**.

First the *nodefs* command is used to extract definitions.

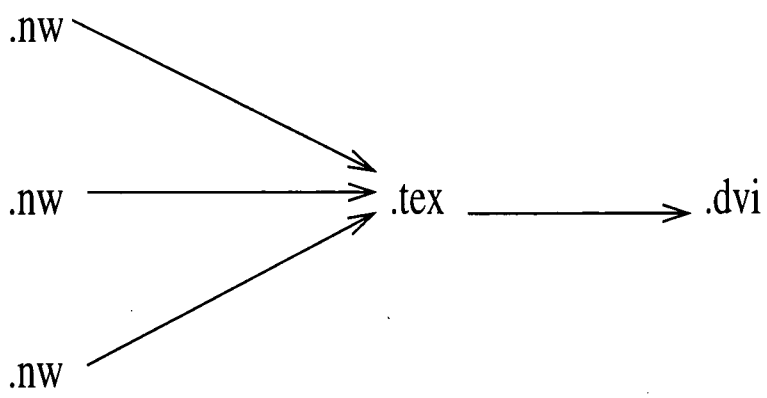

Figure 3.2: many to one

```
nodefs a.nw > a.defs

nodefs b.nw > b.defs

nodefs c.nw > c.defs
```

These are then sorted and piped through cpif as follows:

```
sort -u a.defs b.defs c.defs | cpif all.defs
```

All the identifiers from **a.nw**, **b.nw** and **c.nw** are in **all.defs** sorted in alphabetical order. The following command is used to create a latex file with full cross-reference information for all identifiers.

```
noweave -n -indexfrom all.defs a.nw > a.tex
```

3.3.3 Literate C program

The syntax of a literate C program is as follows:

The root chunk

```
<<*>>=
<<Header files to include>>
```

```
<<Definitions>>
<<Global variables>>
<<The function [[function name]]>>
<<The main program>>
@
```

The root chunk identified by `<<*>>=` contains code chunks and ends with @. The program code identified by the code chunk names is assembled by *noweb* in the same sequence as the code chunks are mentioned in the root chunk. In literate C program, the five high-level code chunks can be defined,

```
<<Header files to include>>,
<<Definitions>>,
<<Global variables>>,
<<The function [[function name]]>>          and
<<The main program>> .
```

There is however, an individual function chunk for each of the functions within the program.

The "header files to include" chunk

`<<Header files to include>>=`

#include "*file-name*"

@

or

`<<Header files to include>>=`

#include <*file-name*>

@

The `<<Header files to include>>=` chunk consists of #include "*file-name*" or #include <*file-name*> and end with @.

The #include preprocessor command causes the entire contents of a specified source text file to be processed as if those contents had appeared in place of the #include command. The #include command has two forms. These are

#include "*file-name*" and #include <*file-name*>.

The form

#include "*file-name*"

typically searches for the file first in the same 'directory' in which the file containing the #include command was found, and then perhaps in other places according to implementation-dependent search rules. However, the form

#include <*file-name*>

typically does not search for the file in the same 'directory' in which the file containing the #include command was found, but only in certain 'standard' places according to implementation-dependent search rules. The general intent is that the "*file-name*" form is used to refer to other files written by the user, whereas the <*file-name* > form is used to refer to standard library files.

In principle an included file may itself contain #include commands. These #include commands are defined in `<<Header files to include>>=` chunk.

The definitions chunk

`<<Definitions>>=`

#define *identifier sequence-of-token*

`@ %def` *identifier*

or

`<<Definitions>>=`

#define *identifier(identifier1,identifier2,....) sequence-of-token*

`@ %def` *identifier identifier1 identifier2 .....*

An *identifier,* also called a *name* in C, is a sequence of letters, digits, and underscores. An identifier must not begin with a digit and it must not have the same spelling as a reserved word. The #define preprocessor command causes an identifier to become defined as a macro to the preprocessor. A sequence of tokens, called the body of the macro, is associated with the name. When the name of the macro is recognised in the program source text or in the documents of certain other preprocessor commands, it is treated as a call to that macro; the name is effectively replaced by a copy of the body. If the macro is defined to accept arguments, then the actual arguments following the macro name are substituted for formal parameters in the macro body.

If there is a single identifier in #define statement, the identifier name is defined as `@ %def` *identifier.* The more complex form of macro definition declares the names of formal parameters within parenthesis, separated by commas. In literate C programs, these macro names and all the argument names which defined after `@ %def` are delimited by space. *noweb* generates a list of identifiers used, where they appear, and also indicates the place of definitions.

The global variables chunk

```
<<Global variables>>=
```

type identifier

@ %def *identifier*

A *type* is a set of values and a set of operations on those values. The C language provides a large selection of built-in types, including integers of several kinds, floating point numbers, pointers, enumerations, arrays, structures, unions and functions. There is also a special type, void, which has no value; it is used to specify functions that return nothing.

In this global variables chunk, the identifiers, used as global variables are declared by its type and identifier name. @ %def line also define the *identifier* used as global variable.

The function chunk

There will be one of these for each function, where [[*function-name*]] is to be replaced by the actual function name.

<<The function [[*function-name*]]>>=

function-name (*parameters*)

parameter declarations

{

<<Variables local to [[*function-name*]]>>

<<Body of [[*function-name*]]>>

}

@ %def *function-name parameters*

<<Variables local to [[*function name*]]>>=

type identifiers

.

@ %def *identifiers*

<<Body of [[*function name*]]>>=

program source code

.

@

In function definitions, formal parameters are declared in two parts. The names of the parameters are listed in the function declarator. In order to supply types for the parameters, the programmer declares each of the parameters (in any order) in the parameter declaration section. For example, to define a function that has three parameters—an integer, a double precision floating point number, and a pointer to an integer— the program can be written as:

```
void f(x,y,z)
  int x, *z;
  double y;
{
  local variables declaration
  body of the function
}
```

In literate C programs, *function-name, parameters and parameter declarations* are written in the <<The function [[*function name*]]>>= chunk. The local variables declaration refers to another chunk <<Variables local to [[*function name*]]>> and the body of the function refers to <<Body of [[*function name*]]>> chunk.

The local variables of the function are declared in <<Variables local to [[*function name*]]>>= chunk. The identifiers of the local variables are defined by @ %def *identifiers* at the end of the chunk.

The program source code of the function is described in the <<Body of [[*function name*]]>>= chunk. This chunk ends with the symbol @.

So for example for the function f above the literate C is :

```
<<The function [[f]]>>=
void f(x,y,z)
  int x, *z;
  double y;
{
<<Variables local to [[f]]>>
<<Body of [[f]]>>
```

```
}
@ %def f x y z

<<Variables local to [[f]]>>=
  int i,j;
  char k;
@ %def i j k

<<Body of [[f]]>>=
.
.
@
```

The main program chunk

This is similar to the function chunk described above, but the main program chunk (described below) is distinguished because it is the entry point to the program.

```
<<The main program>>=
```

main*(parameters)*

parameter declarations

```
{
<<Variables local to [[ main ]]>>
<<Body of [[main]]>>
}
```

@ %def main *parameters*

```
<<Variables local to [[main]]>>=
```

type identifiers

.
.

@ %def *identifiers*

```
<<Body of [[ {\em main} ]]>>=
```

program source code

.
.
@

The <<The main program>>= chunk is the same as the function chunk. *main* and *parameter declarations* are defined in the <<The main program>>= chunk. The local variable declarations and main program source code refer to the chunks <<Variables local to [[main]]>> and <<Body of [[*main*]]>> respectively. The local variables in the main program are declared in <<Variables local to [[main]]>>= chunk and the main program source code is described in the <<Body of [[*main*]]>>= chunk.

3.4 The Capture Process

The general idea of the capture process is to generate literate source code from actual source code. This process is illustrated in Figure 3.3. We assumed that source code could be in any language because *noweb* is a language independent tool but for this application we assume it is in C. The capture process will convert a C program into the form described in the previous section. Obviously no documentation can be captured from the source code, this will have to be added later. The initial result of the capture process can then be processed by *notangle* to generate a C program, and by *noweave* to produce essentially null documentation in TEX form that includes cross references of chunks and identifiers used in the program.

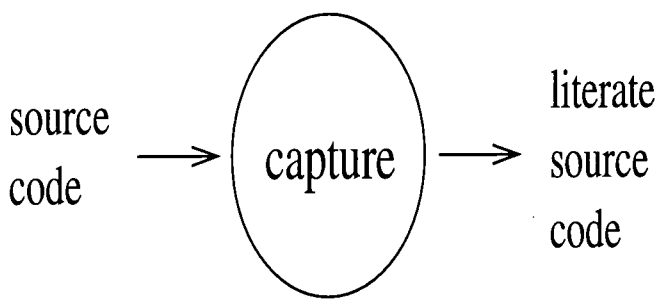

Figure 3.3: Generate a literate source code

The literate program has its own structure as described above, so the program source code is formed into the literate programming structure. Root chunk definition <<*>>= followed by the sequence of chunk names is written into the output file at the beginning of the source code. Then, the definitions of these chunk names are defined by the source code or references of the other chunks. At the end of each code chunk the symbol @ is written in column 1 of the next line. It marks the end of the code chunk or start of the documentation chunk, however there is no documentation chunk. The main program and all the functions or processes or modules of the source code are in each named chunk. The global variables and definitions are also defined in their named chunks.

The generated *noweb* file can be processed by *noweb* tools known as *notangle* and *noweave* as shown in Figure 3.4.

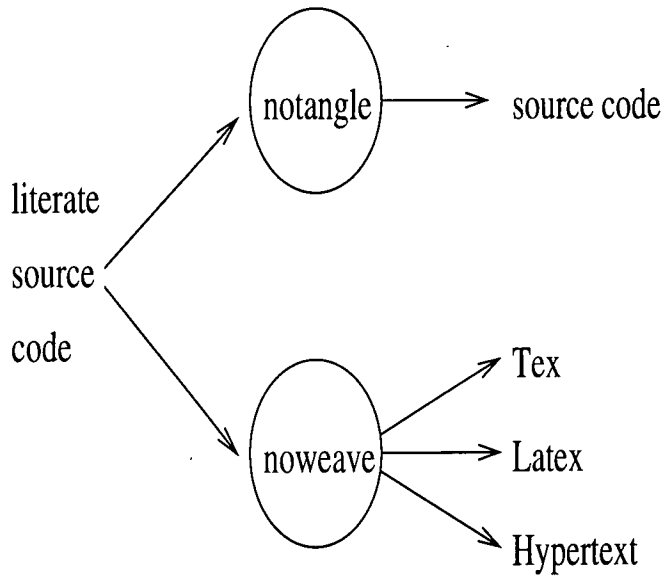

Figure 3.4: Using noweb to build code and documentation

notangle *inputfile.nw* > *outputfile.c*

This command extracts the source code from the *noweb* file (inputfile.nw) file. The source code will be obtained in *outputfile.c* file which can be processed by a C language compiler.

noweave *inputfile.nw* > *outputfile.tex*

The *noweave* command processes the *inputfilename.nw* to *outputfilename.tex*. *noweb* generates cross reference information for chunks and identifiers by using the @ %def markings in its source code.

@ **%def** *identifiers*

@ **%def** *identifier1 identifier2 identifier3 ...*

The above statement means that the preceding chunk defines the identifiers listed in identifiers. This notation provides a way of marking definitions manually when no automatic marking is available. *noweave* can work with LaTeX or it can use a plain TeX macro package, supplied with *noweb*. *noweave* can also work with HTML, the hypertext markup language for Mosaic and the World-Wide Web. Then, *noweave's* final stage is converted to either TeX, LaTeX, or HTML. In the Hypertext system, the information can be accessed through navigation. The cross-reference chunks and identifiers are the nodes and by clicking the mouse on these, the other nodes of the link relation will appear.

When the redocumented programs are stored in a literate source program, the code and documentation can be extracted from the literate C program at any time. To store the existing program in a literate programming style, we need to transform the source code to a literate program. The documentation, produced from the literate program, is the text in the typeset or Hypertext which are cross-referenced. It gives program readable and basic information only. It cannot provide the function of the chunks and other information.

Some of the program analysis tools in a software engineering environment provides useful information about the source code.

The information together with the knowledge gained by a programmer in the process of understanding the program may be stored in the literate program incrementally. The more analysis of the source code, the more information that can be stored. The process of incrementally adding to the literate program form of the original program is described in the next section.

3.5 The Edit Process

There are two activities in the Edit Process, namely View and Change. The View activity is supported by a tool which allows users to browse the views on the screen only and the Change activity is supported by a tool which allows the users to update the literate program.

3.5.1 The View activity

The system is provided with the following five views of the literate program:

- the literate source code,
- the list of code chunks,
- the list of identifiers used,
- the document on hypertext, and
- the program source code.

The literate source code is the generated .nw file which comprises of the code-chunk, the documentation-chunk and the identifier-definitions. When the editor displays the chunk on the screen, the chunk will be shown in a window. The user can move around the screen page by page.

The next view shows the list of code chunks. From this view, the user can see the name of code chunks and the first definition of the chunk is also shown at the end of the chunk name. For example,

<Header files to include 2b>

show that *Header files to include* is the chunk name and it is defined in *2b.*

The third one is the list of identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically. For example,

line_count: 7a, 7b, 8a, 9a, 9b, 10

line_count is defined in chunk 7a, and used in 7b, 8a, 9a, 9b, and 10.

Another view is the document through Hypertext. The document which is redocumented from the program source code will appear on the Hypertext. The related document of the chunks and identifiers can be seen by clicking the mouse on their names.

The last view is the program source code presented by chunk. The user can see the program source code chunk by chunk. At the same time the associate document will appear on the document window.

3.5.2 Change

When some changes are needed in code-chunk the editor allows changes to the codes in code-chunks. It also allows the additions of new chunks and deletions of existing chunks. The associated document can be changed as new information is added. The following change processes can be allowed by the change editor.

1. Modify (including add) documentation of an existing chunk
2. Change source code (to correct a problem or add a new feature)
3. Decomposition of chunks
4. Delete a chunk
5. Composition of chunks

These are described in more detail below.

Modify documentation of an existing chunk

The main purpose of incremental redocumentation is to add information to the existing program as and when knowledge about the program is gained. Hence, the modification of the document or adding new documents to the existing system is essential. The editor will show two windows on the screen, one for literate source code and another for documentation. The literate source code will appear on the code-chunk window by clicking the chunk name on the list of chunks which is browsing on the small window and the associate documentation which will appear on the documentation window. The editor is allowed to modify or add the information on the documentation window. In this way the software engineers can improve upon the existing information and documentation of the programs.

Change source code

When the requirement is changed or the resulting behaviour of the program is not correct, the code needs to be changed. The required code chunk will appear on the code-chunk

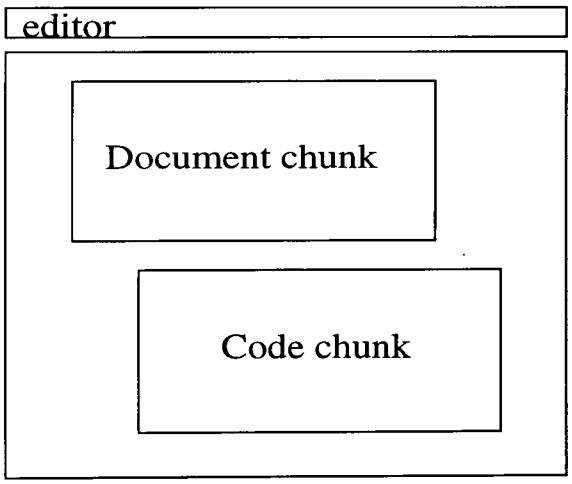

Figure 3.5: Literate program editor

window by clicking the name of code chunk from the listing as above. The system allows the programmer to change the code in code-chunk window. The updated code-chunk is in the *nw* file and the compilable source code can be extracted by *notangle*.

When a change to the source code has been made the programmer will be informed that the related documentation needs to be updated. There are essentially two approaches, the programmer is forced to make a change to the documentation, or an indication of a change can be given. Both approaches have their advantages and disadvantages and the final decision on which to choose will depend on the programmers and the change procedures in place in their working environment.

Decomposition of chunks

The programmer may want to split existing chunks of program source code into smaller chunks. The advantage of this is that smaller chunks are easier to understand and will generally encapsulate one concept. The result from the capture process described above can only be chunks of large granularity which are the bodies of the functions.

The breaking up of an existing chunk *qsort* of the program **lines.nw"** (Appendix G) is shown in Figure 3.6 The code-chunk *qsort* can be selected by clicking the name from the list of code chunks, ie. the first line in Figure 3.6.

The names of chunk to be decomposed are given on the screen. After giving the names, the system automatically writes these chunk names under the existing code chunk as well as in the list of chunks.

The decomposed code chunk can be selected by clicking the code-chunk name from the list of chunks. Next, the code or document can be chosen for code and document windows. If the document is chosen, the document-window will appear and the document of the chunk is shown on the screen. If the code is chosen, similarly the code is shown on the code-window.

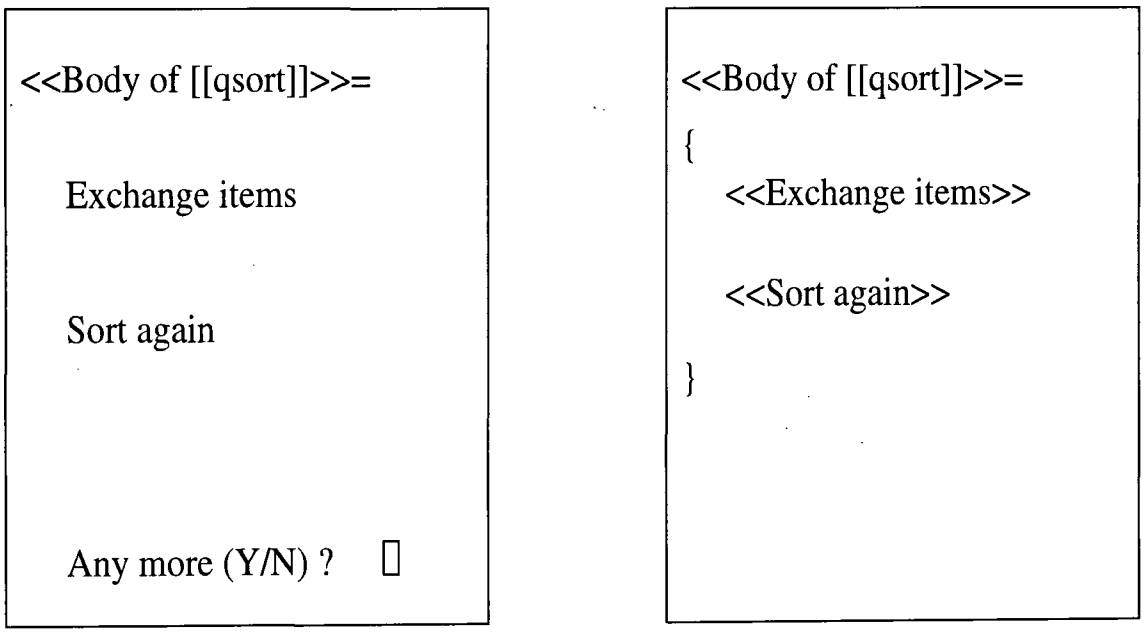

Figure 3.6: Screens before and after breaking up a chunk

The first step is to see if items need to be exchanged in the array `v`.

```
<<Exchange items>>=
y(left >= right)



swap(v,left,last);
@


Now sort the left and right parts of the array separately

<<Sort again>>=
qsort(v,left,last-1);
qsort(v,left+1,right);
@
```

Delete a chunk

There are two kinds of deletion:

1. delete a chunk name and replace it by the associated source code
2. delete all source code associated with a chunk

In the above example, the `<<Sort again>>` chunk is to be deleted. The first process will delete the chunk name `<<Sort again>>` from the `<<Body of [[qsort]]>>=` chunk and that line is replaced by the source code which is written in the `<<Sort again>>=` chunk. The second process will delete the chunk name `<<Sort again>>` from the `<<Body of [[qsort]]>>=` chunk and all the codes which are written in the `<<Sort again>>=` chunk are deleted.

Composition of chunks

Uniting chunks is the reverse form of decomposition. A chunk can be added as a part of another chunk or as an independent chunk. When a new chunk name is introduced it is written in the list of chunk names. There is a simple mechanism for opening a new window to specify this new chunk. The programmer must be able to choose to enter code or text. If it is a code-chunk, the name of the new chunk automatically appears in the window and the programmer will be able to enter code into this new chunk. If it is a document-chunk, the text can be written on the window without names. Some chunks will be added as independent chunks, that is, they are not a part of some other chunks, but form the main chunk for other chunks.

3.6 Example

As mentioned above the capture process generates a literate C program from the C program source code. A literate program *wc.nw* written by Ramsey [41] (Appendix A) is used as an example of *noweb* program. The documents are written by the programmer, so these are not in the fixed format. The chunks can be separated according to the programmer's style. The literate C programs written by the different programmers from the same C program could be different. However, the generated literate C program has the same pattern for all programs.

Here, we will use Ramsey's wc.nw as an example. The C source code file **wc.c** (Appendix B) is processed by the *noweb* command *notangle*, from the literate program **wc.nw**. Then the literate C program **wc.nw'** (Appendix C) is generated from the **wc.c** by the capture process. The stages of the capture process are shown in Figure 3.7.

Now we can see the difference between original **wc.nw** and generated **wc.nw'**. The result **wc.nw'** will never be exactly the same as **wc.nw**. **wc.nw** is written according to the programmer's own idea. Header definition is defined by the programmer in **wc.nw** and there is no need to write in sequence the same header as the other *noweb* programs. For example, the header definition in wc.nw is described as follows.

```
<<*>>=
<<Header files to include>>
<<Definitions>>
<<Global variables>>
```

```
<<Functions>>
<<The main program>>
@
```

But in another literate program it can be defined by the programmer in another way such as changing the sequence of order or writing in different header names. However the root chunk `<<*>>=` and the header definition chunks in **wc.nw'** are generated by the program, so these have the same structure, the same chunk names in every generated literate C program as shown below.

```
<<*>>=
<<Header files to include>>
<<Definitions>>
<<Global variables>>
<<The function [[function name]]>>
<<The main program>>
@
```

The structure of wc.nw' is not similar to wc.nw. In wc.nw, one or more #include can write under `<<Header files to include>>=` chunk heading. For example,

```
<<Header files to include>>=
#include <stdio.h>
#include <string.h>
@
```

But only one #include line can appear under one `<<Header files to include>>=` chunk heading in wc.nw'. If there is more than one #include line, `<<Header files to include>>=` the chunk heading is needed for each #include line.

```
<<Header files to include>>=
#include <stdio.h>
@
<<Header files to include>>=
#include <string.h>
@
```

The definition chunk is similar to the above example, one or more # define is written in wc.nw under `<<Definitions$>>=` chunk.

```
<<Definitions>>=
#define OK               0
  /* status code for successful run */
#define usage_error      1
  /* status code for improper syntax */
#define cannot_open_file  2
  /* status code for file access error */
@ %def OK usage_error cannot_open_file
```

In wc.nw', `<<Definitions>>=` is needed for each # define.

```
<<Definitions>>=
#define OK               0
  /* status code for successful run */
@ %def OK
<<Definitions>>=
#define usage_error      1
  /* status code for improper syntax */
@ %def usage_error
<<Definitions>>=
#define cannot_open_file  2
  /* status code for file access error */
@ %def cannot_open_file
```

The variable definitions and global variables are grouped and presented under the headings `<<Definitions>>=` and `<<Global Variables>>=` respectively by the programmer. The generator can not group them, then each variable is described under each heading as well as `@ %def` line is written for each variable.

The function `wc_print` is described in the `<<Functions>>=` chunk in `wc.nw` as follows:

```
<<Functions>>=
wc_print(which, char_count, word_count, line_count)
  char *which;  /* which counts to print */
  long char_count, word_count, line_count;
    /* given totals */
```

```
{
 ...program source code
}
@ %def wc_print
```

The function chunk is separated into two parts in wc.nw'. These are:

- `<<Variables local to [[wc_print]]>>`; and
- `<<$Body of [[wc_print]]>>`.

Parameters are declared in the `<<The function [[wc_print]]>>=` chunk. Local variables are declared in `<<Variables local to [[wc_print]]>>=` and the program source code is written in `<<$Body of [[wc_print]]>>`.

```
<<The function [[wc_print]]>>=
wc_print(which, char_count, word_count, line_count)

  char *which;  /* which counts to print */
  long char_count, word_count, line_count;
    /* given totals */
{
<<Variables local to [[wc_print]]>>
<<Body of [[wc_print]]>>
}
@ %def wc_print which char_count word_count line_count

<<Variables local to [[wc_print]]>>=
...
@ %def

<<Body of [[wc_print]]>>=

...program source code

@
```

The main program chunk is similar to the function chunk in wc.nw'. It also consists of two chunks `<<Variables local to main>>` and `<<Body of main>>` chunk.

3.7 Summary

This chapter has described a method for incrementally redocumenting a C program by first capturing the source code in the form of a literate C program. Then the captured literate C program can be incrementally redocumented by using the edit process. These steps are illustrated by examples.

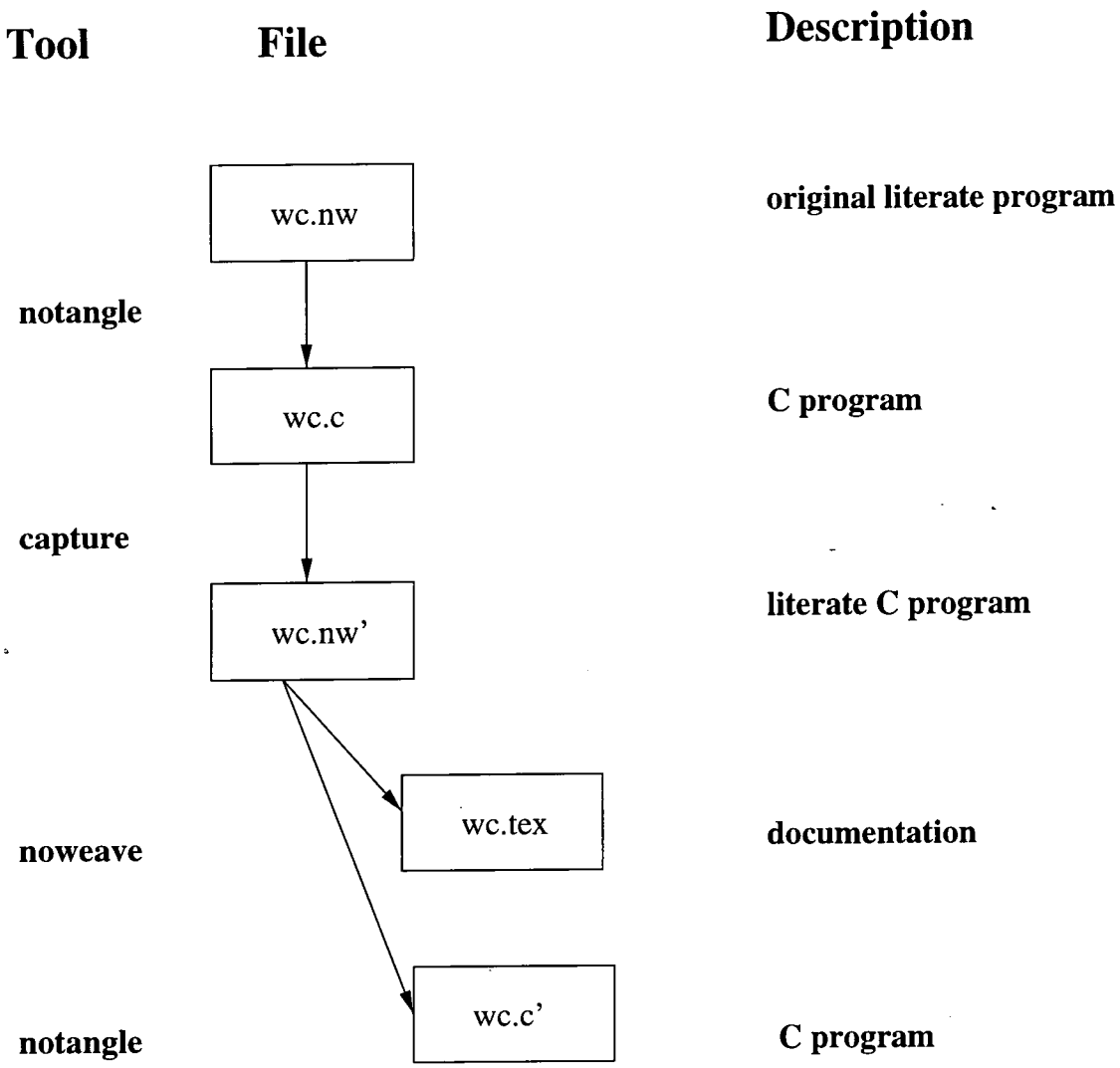

Figure 3.7: Process Carried Out During Example

Chapter 4

Implementation

4.1 Introduction

A literate C program can be generated from existing C source code as mentioned in the previous chapter. In this chapter, the implementation of a tool to support the capture process is described C analysis tools, such as indent, ctags, cxref have been used to generate information from the C source code to use in redocumenting the literate C program 'nw file'. Additional C shell (gen-info), awk (gen-ctags, gen-xref), and C programs (xref-conv, gen-nw) have been developed as part of this research. Here, details of using these tools are described. It was not the intention of this work to develop C analysis tools, so existing tools have been utilised where possible.

4.2 The literate C program generator

In the following description of the literate C program generator the program **wc** is used as an example. The stages of generating the literate C program from the C source code file is shown in Figure 4.1. The commands shown in Figure 4.1 are collected together into a C shell command file called **gen-info**.

The C language allows flexible free format input which could cause problems for analysis tools. So, the generalised format is needed for those C programs. The **indent** tool can be used to form the generalised indentation format of C programs. In Figure 4.1, the *wc.in.c* is the output file produced by the **indent** from the *wc.c*. It reformats the C program according to given options. The options which are specified in the **gen-info** script are described below.

```
indent $cprog $indentedcprog -bap -baac -bad -bbb -nbc -br -cdb -i3 -d1 -di3
```

The above options control the formatting style imposed by indent as follows:

-bap If -bap is specified, a blank line is forced after every procedure body.

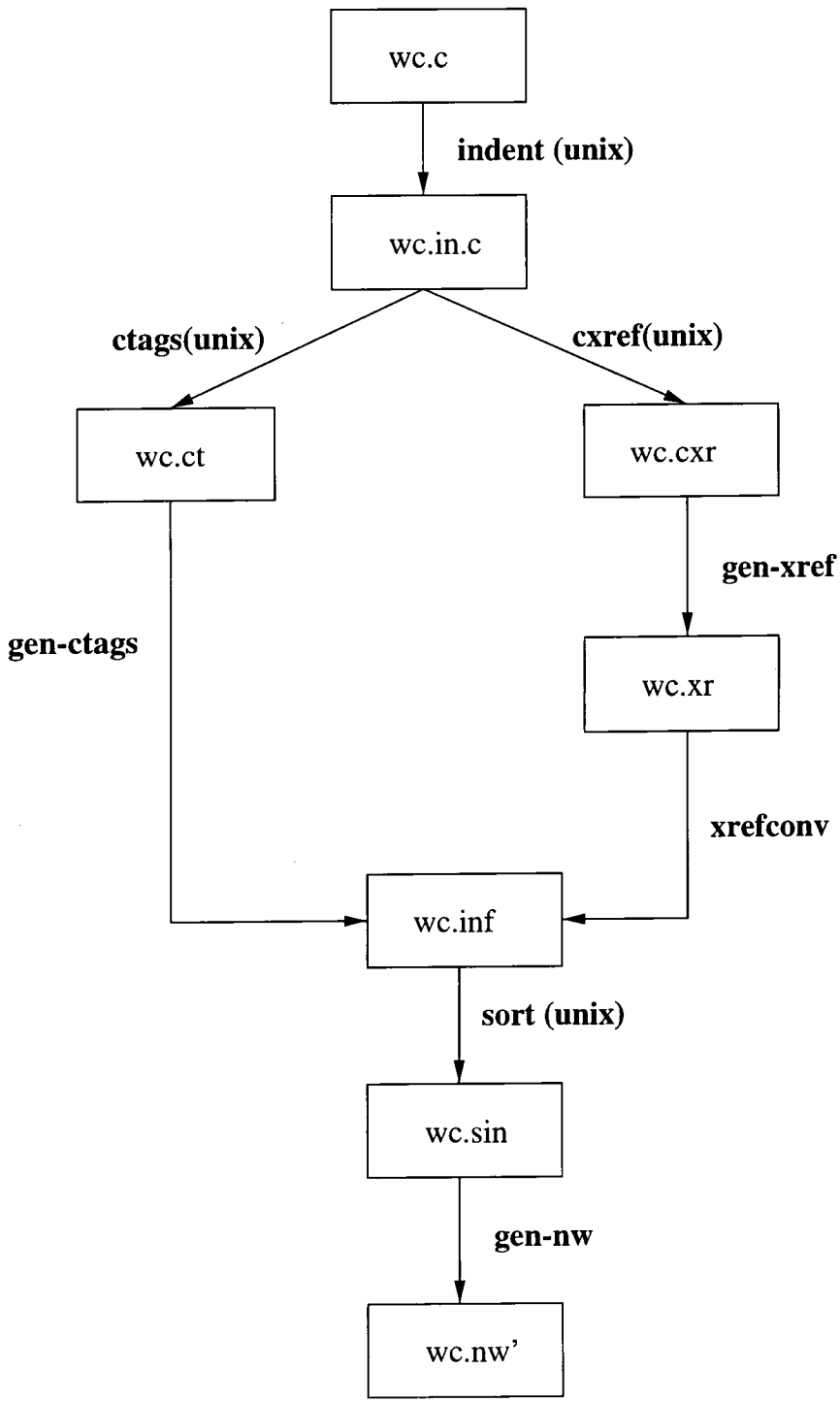

Figure 4.1: Literate C program generator

-bacc If -bacc is specified, a blank line is forced around every conditional compilation block. That is, in front of every #ifdef and after every #endif. Other blank lines surrounding these will be swallowed.

-bad If -bad is specified, a blank line is forced after every block of declarations.

-bbb If -bbb is specified, a blank line is forced before every comment block.

-bc If -bc is specified, then a newline is forced after each comma in a declaration.

-br Specifying -bl lines up compound statements.

-cdb Enables the placement of comment delimiters on blank lines.

-dn Controls the placement of comments which are not to the right of code. The default **-d1** means that such comments are placed one indentation level to the left of the code. Specifying **-d0** lines up these comments with the code.

-din Specifies the indentation, in character positions, from a declaration keyword to the following identifier.

-in The number of spaces for one indentation level.

The indented file,*wc.in.c*, is processed by two different tools, **ctags** and **cxref**. The **ctags** tool makes a tags file from the specified C, Pascal, Fortran, yacc and lex sources. Normally **ctags** places the tag descriptions in a file called tags; this may be overridden with the -f option. By default, the tags file is sorted in lexicographic (ASCII) order. Files with names ending in *.c* or *.h* are assumed to be C source files and are searched for C routine and macro definitions.

In the literate C program generator **gen-info**, **ctags** makes a tags file *wc.ct* from the indented C program source code *wc.in.c* file. The *wc.ct* file is composed of four fields separated by white space, the object name, the line number, the file in which it is defined, and the text of that line. The located line number of the main program and functions can be known by the *wc.ct* file. This file is reformatted by the **gen-ctags** program, the output file *wc.inf* contains `T` character in the first field. `T` indicates for main program or function. The `-twx` options are used in **ctags** command and the functions of these options are described below.

```
ctags -twx $indentedcprogram | sort +2d -3 +1n -2 +0d -1 > $ctagsfile
```

-t Create tags for typedefs

-w Suppress warning diagnostics

-x Produce a list of object names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output.

The file produced by **ctags** is in alphabetical order of the function names and is sorted in order of line number by **sort** command. The sorted file *wc.ct* shows a list of object names, the line number, file name and text of that line in order of line number.

The resulting *wc.ct* is as follows:

```
print_count        12 wc.in.c          #define print_count(n)
printf("%8ld", n)

wc_print           22 wc.in.c          wc_print(which, char_count,
word_count,

main               49 wc.in.c          main(argc, argv)
```

The field positions are reformed by the program **gen-ctags**. The first position is type, and the program writes `T` in all records which indicates the main program or function. The second position indicates the file name, and the function name or main is in third position and the line number shows in fourth position.

The **cxref** tool generates the C program cross reference file *wc.cxr* from the *wc.in.c* file by using the following command in the **gen-info** script.

```
cxref -o $cxreffile $indentedcprogram
```

where

-o file Direct output to named file.

The **cxref** command analyses a collection of C files and builds a cross-reference table. It produces a listing on standard output of all symbols (auto, static, and global) in each individual file, or, with the -c option, in combination. The table includes four fields: object name, file name, function and line number. The line numbers appearing in the line number field also show reference marks as appropriate. The reference marks include:

```
assignment    =
declaration   -
definition    *
```

If no reference marks appear, it can be assumed as a general reference. The located line number of all symbol definitions appear in the *wc.cxr* file by reference marks *.

The program **xrefconv** selects the definition lines from the *wc.cxr* file and the result is stored in *wc.xr* file. The *wc.xr* file is also reformatted by the **gen-xref** program which adds the character *X* in the first position of the output line and appends these lines to the *wc.inf* file. The *wc.inf* file is sorted by **sort** command in order of line number and the result is stored in *wc.sin* file.

```
sort -t  +1d -2 +3n -4  $outfilename >$sortedinfo
```

The *wc.sin* file can be seen as follows:

```
X wc.in.c OK 2
X wc.in.c usage_error 4
X wc.in.c cannot_open_file 6
X wc.in.c READ_ONLY 8
X wc.in.c buf_size 10
T wc.in.c print_count 12
X wc.in.c print_count() 12
X wc.in.c status 13
X wc.in.c prog_name 16
X wc.in.c tot_char_count 19
X wc.in.c tot_line_count 19
X wc.in.c tot_word_count 19
T wc.in.c wc_print 22
X wc.in.c wc_print() 22
X wc.in.c which 23
X wc.in.c char_count 24
X wc.in.c line_count 24
X wc.in.c word_count 24
T wc.in.c main 49
X wc.in.c main() 49
X wc.in.c argc 50
X wc.in.c argv 53
X wc.in.c file_count 57
X wc.in.c which 60
X wc.in.c fd 63
X wc.in.c buffer 66
X wc.in.c ptr 69
X wc.in.c buf_end 72
X wc.in.c c 75
X wc.in.c in_word 78
X wc.in.c char_count 81
X wc.in.c line_count 81
X wc.in.c word_count 81
```

The first character indicates functions as `T` and identifiers as `X`. The **gen-nw** program constructs the literate C program root chunk and header-chunks. The expansions of the chunks are made by analysing the text from the input file *wc.c* which line number is defined in *wc.sin*. The output from the **gen-nw** program is the literate C program *wc.nw'*.

4.3 Summary

This chapter has shown how a literate C program can be generated from a C program by using existing Unix tools and a simple special purpose analysis tool written in C.

For the other programming languages, the analysis tools for each programming language are needed. The C programs in the capture process can be used not only for C programs but also for the other languages.

Chapter 5

Evaluation of the process

5.1 Introduction

In Chapter 3, the structure of the literate C program, the capture process and the edit process was discussed. The previous chapter described the implementation of the capture process and the tools which were utilised. An evaluation of the results is made in this chapter.

5.2 Evaluation

The evaluation of the incremental redocumentation by using literate programming system is to be carried out by comparing example programs. In particular, it will compare Ramsey's literate example program **wc.nw** with the captured and redocumented version. It will also discuss the incremental redocumentation of the **line.c** program.

5.2.1 Wordcount program wc.c

The main idea of the system is to convert a C program into Literate C program by the capture process and then the documentation will have to be added later incrementally. The structures of the captured literate C program **wc.nw'** (Appendix C) and the *noweb* program **wc.nw** (Appendix A) were described in Chapter 3. Obviously the captured literate C program **wc.nw'** cannot achieve the same documentation as Ramsey's literate program **wc.nw**. We are not trying to get the same result as **wc.nw**. **wc.nw** is developed in development phase by top-down stepwise refinement method. In the top-down stepwise refinement method the programmer should define the levels of the program and implements the smaller functions step by step. The *Header Definition* is flexible and it does not need to be exactly the same as the other literate programs.

The programmer can define the *Header Definition* according to their own way of thinking and writing. One chunk name is put in the *Header Definition* and the implementation is

done later.

If there is more than one `#include` line, they are grouped in a `<<Header files to include>>` chunk. `#define` lines are also grouped in a `<<Definitions>>` chunk. The programmer can group the global variables under the `<<Global Variables>>` heading.

While the program is being developed, the main program and the function can be decomposed into smaller chunks and the smaller chunks can be decomposed into even more smaller chunks. There is no limitation for decomposition and it can be described according to the ideas of the programmer. In the literate program **wc.nw** `<<The main program>>` chunk is decomposed into the following smaller chunks.

```
<<Variables local to main>>
<<Set up option selection>>
<<Process all the files>>
<<Print the grand totals if there were multiple files>>
```

The `<<Process all the files>>` chunk is divided into smaller chunks.

```
<<If a file is given, try to open *(++argv); continue if
successful>>
<<Initialize pointers and counters>>
<<Scan file>>
<<Write statistics for file>>
<<Close file>>
<<Update grand totals>>
```

These chunks can also be decomposed if the programmer wants to do so. In the **wc.nw**, the programmer writes the information about the literate noweb program wc.nw before the header definition. The information about the code chunk is described in the documentation chunk. These documentation chunks were written in the development phase and described what the programmer wanted to do while the program was being developed. They do not contain information about the analysis of the program.

However, the captured literate C program has only the C code structure as a literate program. It is developed in the maintenance phase of the software life-cycle when the program already exists. In our example, the literate C program **wc.nw'** has fixed program structure because it is generated by the capture process. The program structure of all captured programs will be exactly the same structure except for the number of functions and the function names. The `#include` lines are not grouped together under a `<<Header files to include>>` header, so it is different from the **wc.nw**. There is a `<<Header files to include>>` header for each `#include` line. The `#define` and `global variables` are also the same as `#include`. They are not grouped together under `<<Definitions>>` and `<<Global variables>>`. For each `#define` line, there will be a header `<<Definitions>>` and for each global variable, there will be a header `<<Global variables>>`. So, we can say that there are no groupings of the `<<Header files to include>>`, `<<Definitions>>`, and `<<Global variables>>`. The

individual chunks are created for each individual line. The function chunks are defined as `<<The function [[name]]>>`. Two smaller chunks `<<Variables local to [[name]]>>` and `<<Body of [[name]]>>` are generated under a function chunk. The first is to define the local variables and the latter contains the program source code of the function. These two chunks cannot be decomposed into smaller chunks in the literate C program **wc.nw** because these are automatically generated by the capture process. `<<The main program>>` chunk has the same structure, two chunks `<<Variables local to main>>` and `<<Body of main>>`. These chunks also cannot be decomposed. It has no information about the program and there will be no documentation chunk. But from this captured literate C program we can get the cross- reference information of code chunks and identifiers as the literate program **wc.nw**. The resulting output includes cross-reference information for identifiers and list of the code chunks.

The captured literate C program **wc.nw'** is then changed to redocument it as a literate C program **wc.nw"** (Appendix D). The documentation for the appropriate code chunk relies initially on the comment lines contained within the source code. If there are no comment lines in existing source code, no documentation will appear in the preceding documentation chunk. So, there is a question, *What do we do with comments*? The answer is: first, to tie comments to code; which means that it is to find out which comment is related to which code. Then, the second step is to convert these comments to documentation. The comment lines in source code are placed together in the documentation chunk by using the **Edit process**. If we want to add more chunks, the new chunk names can be added to the *Header Definition*. And also, a chunk could be delete from the *Header Definition* when it needs to be deleted. Under the `<<Header files to include>>` chunk, the `#include` lines could be possibly be grouped. Similarly, the `#define` and `global variables` could possibly be grouped under the `<<Definitions>>` and `<<Global Variables>>` chunks. The main program and the function chunks could possibly be made smaller by further decomposition according to program understanding. The information from program analysis tools can be added to the documentation chunks. Then the redocumented literate C program **wc.nw"** is more understandable than the captured literate C program **wc.nw'**. **wc.nw"** has not only program source code but also partial documentation. Of course, the redocumented literate C program **wc.nw"** which result is not exactly the same as Ramsey's **wc.nw** file, but it is clear that the printed output document is more understandable than the C source code **wc.c** file.

5.2.2 Lines program lines.c

To confirm this statement, we choose another C program **lines.c** (Appendix E) as an example. The **lines.c** has no documentation and it has source code only. The **lines.c** is captured to the literate C program **lines.nw'** (Appendix F). The captured literate C program **lines.nw'** is redocumented and the result is in **lines.nw"** (Appendix G). The printed output is generated by LaTeX and the result is shown in (Appendix H). The header definition is shown as follows:

```
<*>=
```

```
<Headerfiles to include>
<Definitions>
<Global Variables>
<The function alloc>
<The function getline>
<The function readline>
<The function writeline>
<The function swap>
<The function qsort>
<The main program>
```

When reading this header definition, the reader can understand the structure of the program, and the expansion of these header definitions can be seen clearly. The cross-reference information for identifiers are given in each code chunk. It defines the variables, which variable is used in which chunk, and also gives the information for the code chunk which is used in which chunk. The example of the expansion of the code chunk `<<The function qsort>>` is shown below.

```
6a   <The function qsort 6a>=
       qsort(v, left, right)
       char *v[];
       int left, right;
       {
       <Variables local to qsort 6b>
       <Body of qsort 6c>
       }
     Defines:
       left, used in chunk 6c.
       qsort, used in chunk 6c and 7b.
       right, used in chunk 6c.
       v, used in chunks 5c and 6c.
     This code is used in chunk 1a.

6b   <Variables local to qsort 6b>=
       int i, last;
     Defines:
       i, used in chunks 3c,5e and 6c.
       last, used in chunk 6c.
     This code is used in chunk 6a.
```

The cross-reference for identifiers and code chunks are useful for software engineers to understand the variables and interrelation between chunks. The understanding of the variables used is important to software engineers. Cross-referencing of chunks and identifiers makes

large programs easier to understand. The printed output of the sample program **lines.nw'** shows full cross-reference information. The page numbers are used as a cross-reference. A list of code chunks and the Index are described at the end of the document as shown in the (Appendix F). The first definition of the chunk is described at the end of the code chunk. For example,

```
<Body of alloc 2e>
```

shows that the code chunk `<<Body of alloc>>` can be seen in page number 2. There is more than one code chunk on this page and these are distinguished by a letter. The `<<Body of alloc>>` chunk is described in e chunk. The Index shows a list of identifiers used and where they appear. For example,

`allocbuf`: 1g, 2a, 2e

shows that the identifier `allocbuf` is defined in `1g` and `2a` and used in 2e. So it is more understandable to the software engineers about chunks and identifiers used. There is no documentation chunks in this example, but the software engineers can get more useful information than source code only. They can get the typeset documentation without knowing about LaTeX . When the software engineers gain more knowledge about the program by using analysis tools or any other way, it can be added to the documentation chunks of the literate C program file by the edit process.

This provides incremental support for redocumentation. Here we can see the redocumented literate C program **lines.nw"**. As mentioned above, there is no documentation chunk in the captured literate C program **lines.nw'**. First, we are looking for the comment lines which are related to the appropriate code chunk. Then these comment lines are moved to the documentation chunk. In **lines.nw**,

```
<<Definitions>>=
#define MAXLINES 10 /* max #lines to be sorted */
```

is changed in **lines.nw"** as below.

```
@
MAXLINES is the max #lines to be sorted
<<Definitions>>=
#define MAXLINES 10
@
```

The function chunk `<<Body of [[qsort]]>>` is decomposed into smaller chunks. The following is the `<<Body of [[qsort]]>>` chunk of **lines.nw'** program.

```
<<Body of [[qsort]]>>=
if(left >= right)
```

```
  return;
swap(v, left, (left+right)/2);
last = left;
for ( i=left+1; i <+ right; i++)
    if (strcmp(v[i], v[left]) < 0)
       swap (v, ++last, i);
swap(v, left, last);
qsort(v, left, last-1);
qsort(v, last+1, right);
```

The <<Body of [[qsort]]>> chunk is decomposed into

- <<Exchange items>> and
- <<Sort again>> chunks.

It can be written in the program **lines.nw**" as

```
<<Body of [[qsort]]>>=

<<Exchange items>>
<<Sort again>>
@
```

The <<Sort again>> chunk is decomposed into more smaller chunks

- <<Sort left part of the array>> and
- <<Sort right part of the array>> .

```
<<Sort again>>=
for ( i=left+1; i <+ right; i++)
    if (strcmp(v[i], v[left]) < 0)
       swap (v, ++last, i);
swap(v, left, last);

<<Sort left part of the array>>
<<Sort right part of the array>>
@
```

The declaration of the <<Sort left part of the array>> can be written in

```
<<Sort left part of the array>>=
qsort(v, left, last-1);
@
```

And also the <<Sort right part of the array>> can be declared as

```
<<Sort right part of the array>>=
qsort(v, last+1, right);
@
```

After decomposition of the <<Body of [[qsort]]>> chunk can be seen as follows. The number of chunks will be increased but the smaller chunks can be more understandable than the large chunk.

```
<<Body of [[qsort]]>>=

<<Exchange items>>
<<Sort again>>
@

<<Exchange items>>=
if(left >= right)
  return;
swap(v, left, (left+right)/2);
last = left;
@
<<Sort again>>=
for ( i=left+1; i <+ right; i++)
    if (strcmp(v[i], v[left]) < 0)
       swap (v, ++last, i);
swap(v, left, last);

<<Sort left part of the array>>
<<Sort right part of the array>>
@

<<Sort left part of the array>>=
qsort(v, left, last-1);
@
<<Sort right part of the array>>=
qsort(v, last+1, right);
@
```

The decomposition of the chunks has been described. If the composition of the chunks is needed, the software engineer can do the composition of the chunks according to the program understanding. The other information can be added. In this program, the call graph, drawn by ccg tool, is added to the document. At last, the captured C program **lines.nw** and the redocumented captured literate C program **lines.nw"** have the same source code but totally different in style and documentation.

5.3 Summary

In summary, first we compare Ramsey's literate program **wc.nw** and our captured redocumented literate C program **wc.nw"**. The captured redocumented literate C program is not the same as the literate program which is written by the programmer. But where no literate form of the program has been produced during development, through the captured redocument literate program, the software engineers get the advantages of literate programming, such as storing the program source code and documentation together and also they get the advantages of LaTeX such as typeset pretty printing documents, without learning about literate programming and the LaTeX tool.

The captured redocumented literate C program is more readable and understandable than source code only, and it provides the basis for subsequent maintenance and further redocumentation.

Chapter 6

Conclusions

6.1 Introduction

Documentation systems for software are in the main aimed at the development phase of the software life cycle. We have identified some desirable features of a documentation system for software maintenance to support some of the processes involved in redocumentation and use of documentation in maintenance, together with some potential problem areas. Program comprehension is one of the most important activities in software maintenance and reuse. Documentation is needed for program understanding. Some of the existing systems are without documentation or have poor documentation.

Using the methods and tools developed here, a new documentation can be generated for an existing system from source code. In response to a maintenance request, the existing system may be (wholly or more probably partially) redocumented to understand the system.

6.2 The Achievements of the Criteria for Success

The criteria for the success of this thesis have been described in Chapter 1 are as follows:

1. description of documentation methods
2. exploration of literate programming
3. development of a method for redocumenting C programs and literate C programs
4. development of prototype tools to capture literate C programs
5. description of the redocumentation (edit) process

The description of documentation methods was described in Chapter 2. Some problems of poor documentation or without documentation for an existing system have been described.

Types of documentation, such as development documentation, user documentation and technical documentation are useful sources for maintainers and have been discussed.

The exploration of literate programming also have been described in Chapter 2. Literate programming is a tool for development documentation. The detail structure of a literate programming tool **noweb** has been described. Where documentation does not exist, it will be necessary to recreate it by examination of the source code. The redocumentation methods have also been discussed.

In chapter 3, the concept of the incremental redocumentation was described. The features of the C program and the literate program noweb were described. The difference between C program and literate C program have been discussed. The captured literate C program is generated by the capture process from the C source code. The development of the capture process has been described in detail. The documentation is developed from the captured C program by using **noweb** tool, **noweave**.

The redocumentation process of the captured literate C program is done by the edit process. The following change processes has been done by the change editor.

1. Modify (including add) documentation of an existing chunk
2. Change source code (to correct a problem or add a new feature)
3. Decomposition of chunks
4. Delete a chunk
5. Composition of chunks

The detailed process of these have been illustrated by examples. In the example section of the chapter has discussed the difference between Ramsey's literate program **wc.nw** and the captured literate C program wc.nw'.

In Chapter 4, the implementation of the capture process and the development of prototype tools to capture literate C programs has been described. The C analysis tools, indent, ctags, cxref has been illustrated to generate the literate C program. The C shell, awk, and C programs which were also developed and utilized have been described. The C analysis tools, such as indent, ctags, cxref are used to generate information from C source code to the literate C program 'nw file'. C shell, awk, and C programs are also used. Here, details of using these tools are described.

The system has been evaluated from the point of view of the redocumentation in Chapter 5. The difference between an existing literate C program and the captured literate C program has been illustrated by the word count program **wc.nw** and **wc.nw'**. The captured C program **wc.nw'** is not the same as the literate program **wc.nw** but the system allows the redocumentation process using the edit process. This capture process have been illustrated by the redocumented captured literate C program **lines.nw"** (See appendix G.) The comment lines from the source code have moved to the documentation chunks. Some include lines, definitions, global variables have been grouped. The function chunks have been decomposed into smaller chunks in program lines.nw".

6.3 Future Research

6.3.1 Implementation of Edit process

The idea of the edit process has been described in Chapter 3, 4 and 5. In this thesis, the editor emacs has been used to illustrate the examples. The implementation of the edit process is obviously needed. The redocumentation process needs the editor to modify the program source code and documentations by using separate screens. The composition and decomposition functions also need the editor.

6.3.2 Movements of the Comment lines

As illustrated in Chapter 5, the comment lines should be moved to the documentation chunks automatically. The movements have been done manually in the examples, but it could be done automatically by implementing a program in future. When a comment line exists, it will be changed to the documentation line in the documentation chunk automatically.

6.3.3 Grouping process

The grouping process of the include lines, definitions and global variables could be done automatically. The program should decide how many lines to be grouped. All the include lines, definitions and global variables should be grouped or not, at the request of the redocumentation system user.

6.3.4 Analysis Tools for Other Programming Languages

To capture the source code to the literate program the analysis tools for each programming language are needed. Only tools for C were investigated here.

6.4 Summary

This thesis has described a method for incrementally redocumenting a C program by first capturing the source code in the form of a literate C program. The captured literate C program has cross-reference for chunks and identifiers and also printed type set documentation by using LaTeX.

When software engineers read programs, they can understand the behaviour of the code chunks by using their knowledge and experience. The gained understanding can be added to the documentation chunks and these documents are always tied together with the code chunks. If the program is retrieved many times by different software engineers for the

purpose of maintenance or reuse, the different views of the documentation will be added to the documentation chunk. This documentation cannot be lost and can be seen on the computer screen or as a typeset document report or in a book form for the future. It will reduce the time for program comprehension in the software maintenance and reuse phase of the software life cycle.

This chapter has presented the main achievements of the research, the general conclusion and some suggestions for future research.

Appendix A

wc.nw

```
@
Most literate C programs share a common structure.
It's probably a good idea to state the overall structure explicitly at
the outset, even though the various parts could all be introduced in
chunks named \LA{*}\RA{} if we wanted to add them piecemeal.

Here, then, is an overview of the file {\tt wc.c} that is defined by
the {\tt noweb} program {\tt wc.nw}:
<<*>>=
<<Header files to include>>
<<Definitions>>
<<Global variables>>
<<Functions>>
<<The main program>>
@
We must include the standard I/O definitions, since we want to send
formatted output to [[stdout]] and [[stderr]].
<<Header files to include>>=
#include <stdio.h>
@

The [[status]] variable will tell the operating system if the run was
successful or not, and [[prog_name]] is used in case there's an error
message to be printed.
<<Definitions>>=
#define OK              0
  /* status code for successful run */
#define usage_error     1
  /* status code for improper syntax */
#define cannot_open_file 2
  /* status code for file access error */
```

```
@ %def OK usage_error cannot_open_file
<<Global variables>>=
int status = OK;
  /* exit status of command, initially OK */
char *prog_name;
  /* who we are */
@ %def status prog_name

Now we come to the general layout of the [[main]]
function.
<<The main program>>=
main(argc, argv)
  int argc;
    /* number of arguments on UNIX command line */
  char **argv;
    /* the arguments, an array of strings */
{
  <<Variables local to [[main]]>>
  prog_name = argv[0];
  <<Set up option selection>>
  <<Process all the files>>
  <<Print the grand totals if there were multiple files>>
  exit(status);
}
@ %def main argc argv

If the first argument begins with a '{\tt-}', the
user is choosing the desired counts and specifying
the order in which they should be displayed.
Each selection is given by the
initial character (lines, words, or characters).
For example, '{\tt-cl}' would cause just the
number of characters and the number of lines to
be printed, in that order.

We do not process this string now; we simply remember where it is.
It will be used to control the formatting at output time.
<<Variables local to [[main]]>>=
int file_count;
  /* how many files there are */
char *which;
  /* which counts to print */
@ %def file_count which
<<Set up option selection>>=
which = "lwc";
  /* if no option is given, print 3 values */
if (argc > 1 && *argv[1] == '-') {
```

```
  which = argv[1] + 1;
  argc--;
  argv++;
}
file_count = argc - 1;
@
```

Now we scan the remaining arguments and try to open a file, if possible.
The file is processed and its statistics are given.
We use a [[do ... while]] loop because we should read from the standard
input if no file name is given.

```
<<Process all the files>>=
argc--;
do {
  <<If a file is given, try to open [[*(++argv)]];
    [[continue]] if unsuccessful>>
  <<Initialize pointers and counters>>
  <<Scan file>>
  <<Write statistics for file>>
  <<Close file>>
  <<Update grand totals>>
    /* even if there is only one file */
} while (--argc > 0);
@
```

Here's the code to open the file. A special trick allows us to handle
input from [[stdin]] when no name is given.
Recall that the file descriptor to [[stdin]] is~0; that's what we use
as the default initial value.

```
<<Variables local to [[main]]>>=
int fd = 0;
  /* file descriptor, initialized to stdin */
@ %def fd
<<Definitions>>=
#define READ_ONLY 0
  /* read access code for system open */
@ %def READ_ONLY
<<If a file is given, try to open [[*(++argv)]];
   [[continue]] if unsuccessful>>=
if (file_count > 0
&& (fd = open(*(++argv), READ_ONLY)) < 0) {
  fprintf(stderr,
    "%s: cannot open file %s\n",
    prog_name, *argv);
  status |= cannot_open_file;
  file_count--;
  continue;
```

```
}
<<Close file>>=
close(fd);
@
We will do some homemade buffering in order to speed things up:
Characters will be read into the [[buffer]] array before we process
them.
To do this we set up appropriate pointers and counters.
<<Definitions>>=
#define buf_size BUFSIZ
  /* stdio.h BUFSIZ chosen for efficiency */
@ %def buf_size
<<Variables local to [[main]]>>=
char buffer[buf_size];
  /* we read the input into this array */
register char *ptr;
  /* first unprocessed character in buffer */
register char *buf_end;
  /* the first unused position in buffer */
register int c;
  /* current char, or # of chars just read */
int in_word;
  /* are we within a word? */
long word_count, line_count, char_count;
  /* # of words, lines, and chars so far */
@ %def buffer ptr buf_end in_word word_count line_count char_count
<<Initialize pointers and counters>>=
ptr = buf_end = buffer;
line_count = word_count = char_count = 0;
in_word = 0;
@

The grand totals must be initialized to zero at the beginning of the
program.
If we made these variables local to [[main]], we would have to do this
initialization explicitly; however, C's globals are automatically
zeroed. (Or rather, ``statically zeroed.'')  (Get it?)
<<Global variables>>=
long tot_word_count, tot_line_count,
     tot_char_count;
  /* total number of words, lines, chars */
@

\vskip0pt plus3in\penalty-500\vskip0pt plus-3in
The present chunk, which does the counting that is {\tt wc}'s {\em
raison d'\^etre}, was actually one of the simplest to write.
We look at each character and change state if it begins or ends a word.
```

```
<<Scan file>>=
while (1) {
  <<Fill [[buffer]] if it is empty; [[break]] at end of file>>
  c = *ptr++;
  if (c > ' ' && c < 0177) {
    /* visible ASCII codes */
    if (!in_word) {
      word_count++;
      in_word = 1;
    }
    continue;
  }
  if (c == '\n') line_count++;
  else if (c != ' ' && c != '\t') continue;
  in_word = 0;
    /* c is newline, space, or tab */
}
@
```

Buffered I/O allows us to count the number of characters almost for free.

```
<<Fill [[buffer]] if it is empty; [[break]] at end of file>>=
if (ptr >= buf_end) {
  ptr = buffer;
  c = read(fd, ptr, buf_size);
  if (c <= 0) break;
  char_count += c;
  buf_end = buffer + c;
}
@
```

It's convenient to output the statistics by defining a new function [[wc_print]]; then the same function can be used for the totals. Additionally we must decide here if we know the name of the file we have processed or if it was just [[stdin]].{\hfuzz=11.1pt\par}

```
<<Write statistics for file>>=
wc_print(which, char_count, word_count,
         line_count);
if (file_count)
  printf(" %s\n", *argv); /* not stdin */
else
  printf("\n");           /* stdin */
@
<<Update grand totals>>=
tot_line_count += line_count;
tot_word_count += word_count;
tot_char_count += char_count;
```

```
@

We might as well improve a bit on {\sc Unix}'s {\tt wc} by displaying
the number of files too
<<Print the grand totals if there were multiple files>>=
if (file_count > 1) {
  wc_print(which, tot_char_count,
           tot_word_count, tot_line_count);
  printf(" total in %d files\n", file_count);
}
@

Here now is the function that prints the values according to the
specified options.
The calling routine is supposed to supply a newline.
If an invalid option character is found we inform the user about proper
usage of the command.
Counts are printed in 8-digit fields so that they will line up in
columns.
<<Definitions>>=
#define print_count(n) printf("%8ld", n)
@ %def print_count
<<Functions>>=
wc_print(which, char_count, word_count, line_count)
  char *which;  /* which counts to print */
  long char_count, word_count, line_count;
    /* given totals */
{
  while (*which)
    switch (*which++) {
      case 'l': print_count(line_count);
        break;
      case 'w': print_count(word_count);
        break;
      case 'c': print_count(char_count);
        break;
      default:
        if ((status & usage_error) == 0) {
          fprintf(stderr,
            "\nUsage: %s [-lwc] [filename ...]\n",
            prog_name);
          status |= usage_error;
        }
    }
}
@ %def wc_print
Incidentally, a test of this program against the system {\tt wc}
```

```
command on a SPARCstation showed that the ``official'' {\tt wc} was
slightly slower.
Furthermore, although that {\tt wc} gave an appropriate error message
for the options `{\tt-abc}', it made no complaints about the options
`{\tt-labc}'!
Dare we suggest that the system routine might have been better if its
programmer had used a more literate approach?
@
\section*{List of code chunks}
This list is generated automatically.
The numeral is that of the first definition of the chunk.
\nowebchunks
\begin{multicols}{2}[\section*{Index}
Here is a list of the identifiers used, and where they appear.
Underlined entries indicate the place of definition.
This index is generated automatically.]
\nowebindex
\end{multicols}
```

Appendix B

wc.c

```
#include <stdio.h>
#define OK              0
  /* status code for successful run */
#define usage_error     1
  /* status code for improper syntax */
#define cannot_open_file 2
  /* status code for file access error */
#define READ_ONLY 0
  /* read access code for system open */
#define buf_size BUFSIZ
  /* stdio.h BUFSIZ chosen for efficiency */
#define print_count(n) printf("%8ld", n)
int status = OK;
  /* exit status of command, initially OK */
char *prog_name;
  /* who we are */
long tot_word_count, tot_line_count,
     tot_char_count;
  /* total number of words, lines, chars */
wc_print(which, char_count, word_count, line_count)
  char *which;  /* which counts to print */
  long char_count, word_count, line_count;
    /* given totals */
{
  while (*which)
    switch (*which++) {
      case 'l': print_count(line_count);
        break;
      case 'w': print_count(word_count);
        break;
```

```
      case 'c': print_count(char_count);
        break;
      default:
        if ((status & usage_error) == 0) {
          fprintf(stderr,
            "\nUsage: %s [-lwc] [filename ...]\n",
            prog_name);
          status |= usage_error;
        }
    }
}
main(argc, argv)
  int argc;
    /* number of arguments on UNIX command line */
  char **argv;
    /* the arguments, an array of strings */
{
  int file_count;
    /* how many files there are */
  char *which;
    /* which counts to print */
  int fd = 0;
    /* file descriptor, initialized to stdin */
  char buffer[buf_size];
    /* we read the input into this array */
  register char *ptr;
    /* first unprocessed character in buffer */
  register char *buf_end;
    /* the first unused position in buffer */
  register int c;
    /* current char, or # of chars just read */
  int in_word;
    /* are we within a word? */
  long word_count, line_count, char_count;
    /* # of words, lines, and chars so far */
  prog_name = argv[0];
  which = "lwc";
    /* if no option is given, print 3 values */
  if (argc > 1 && *argv[1] == '-') {
    which = argv[1] + 1;
    argc--;
    argv++;
  }
  file_count = argc - 1;
  argc--;
  do {
    if (file_count > 0
```

```
  && (fd = open(*(++argv), READ_ONLY)) < 0) {
    fprintf(stderr,
      "%s: cannot open file %s\n",
      prog_name, *argv);
    status |= cannot_open_file;
    file_count--;
    continue;
  }
  ptr = buf_end = buffer;
  line_count = word_count = char_count = 0;
  in_word = 0;
  while (1) {
    if (ptr >= buf_end) {
      ptr = buffer;
      c = read(fd, ptr, buf_size);
      if (c <= 0) break;
      char_count += c;
      buf_end = buffer + c;
    }
    c = *ptr++;
    if (c > ' ' && c < 0177) {
      /* visible ASCII codes */
      if (!in_word) {
        word_count++;
        in_word = 1;
      }
      continue;
    }
    if (c == '\n') line_count++;
    else if (c != ' ' && c != '\t') continue;
    in_word = 0;
      /* c is newline, space, or tab */
  }
  wc_print(which, char_count, word_count,
           line_count);
  if (file_count)
    printf(" %s\n", *argv); /* not stdin */
  else
    printf("\n");           /* stdin */
  close(fd);
  tot_line_count += line_count;
  tot_word_count += word_count;
  tot_char_count += char_count;
    /* even if there is only one file */
} while (--argc > 0);
if (file_count > 1) {
  wc_print(which, tot_char_count,
```

```
            tot_word_count, tot_line_count);
    printf(" total in %d files\n", file_count);
  }
  exit(status);
}
```

Appendix C

wc.nw'

The literate C program **wc.nw'** is captured from the C program **wc.c**. The result is as follows:

```
<<*>>=
<<Header files to include>>
<<Definitions>>
<<Global Variables>>
<<The function [[ wc_print ]]>>
<<The function [[ wc_print ]]>>
<<The main program>>
@
<<Header files to include>>
#include <stdio.h>
@

<<Definitions>>=
#define OK              0
/* status code for successful run */
@ %def OK
<<Definitions>>=
#define usage_error     1
/* status code for improper syntax */
@ %def usage_error
<<Definitions>>=
#define cannot_open_file 2
/* status code for file access error */
@ def% cannot_open_file
<<Definitions>>=
#define READ_ONLY 0
/* read access code for system open */
```

```
@ %def READ_ONLY
<<Definitions>>=
#define buf_size BUFSIZ
/* stdio.h BUFSIZ chosen for efficiency */
@ def% buf_size BUFSIZ
<<Functions>>=
#define print_count(n) printf("%8ld", n)
@ %def print_count
int             status = OK;
/* exit status of command, initially OK */
char           *prog_name;
/* who we are */
long            tot_word_count, tot_line_count, tot_char_count;
/* total number of words, lines, chars */
<<The function [[ wc_print ]]>>=
wc_print(which, char_count, word_count, line_count)
        char           *which;  /* which counts to print */
        long            char_count, word_count, line_count;
/* given totals */
{

<<Variables local to [[ wc_print ]]>>

<<Body of [[ wc_print ]]>>

}

@ %def wc_print which char_count word_count line_count

<<Variables local to [[ wc_print ]]>>=


@ %def

<<Body of [[ wc_print ]]>>=


        while (*which)
                switch (*which++) {
                case 'l':
                        print_count(line_count);
                        break;
                case 'w':
                        print_count(word_count);
                        break;
                case 'c':
                        print_count(char_count);
```

```
                        break;
                default:
                        if ((status & usage_error) == 0) {
                                fprintf(stderr,
                                        "\nUsage: %s [-lwc] [filename ...]\n",
                                          prog_name);
                                status |= usage_error;
                        }
                }
```

@

<<The main program>>=

```
main(argc, argv)
        int             argc;
/* number of arguments on UNIX command line */
        char          **argv;
/* the arguments, an array of strings */
{

<<Variables local to [[ main ]]>>

<<Body of [[ main ]]>>

}
```

@ %def main argc argv

<<Variables local to [[main]]>>=

```
        int             file_count;
        /* how many files there are */
        char           *which;
        /* which counts to print */
        int             fd = 0;
        /* file descriptor, initialized to stdin */
        char            buffer[buf_size];
        /* we read the input into this array */
        register char  *ptr;
        /* first unprocessed character in buffer */
        register char  *buf_end;
        /* the first unused position in buffer */
        register int    c ;
        /* current char, or # of chars just read */
        int             in_word;
        /* are we within a word? */
```

```
        long            word_count, line_count, char_count;
        /* # of words, lines, and chars so far */

@ %def file_count which fd buffer buf_size ptr buf_end c
@ %def in_word word_count line_count char_count

<<Body of [[ main ]]>>=

        prog_name = argv[0];
        which = "lwc";
        /* if no option is given, print 3 values */
        if (argc > 1 && *argv[1] == '-') {
                which = argv[1] + 1;
                argc--;
                argv++;
        }
        file_count = argc - 1;
        argc--;
        do {
                if (file_count > 0
                    && (fd = open(*(++argv), READ_ONLY)) < 0) {
                        fprintf(stderr,
                                "%s: cannot open file %s\n",
                                prog_name, *argv);
                        status |= cannot_open_file;
                        file_count--;
                        continue;
                }
                ptr = buf_end = buffer;
                line_count = word_count = char_count = 0;
                in_word = 0;
                while (1) {
                        if (ptr >= buf_end) {
                                ptr = buffer;
                                c = read(fd, ptr, buf_size);
                                if (c <= 0)
                                        break;
                                char_count += c;
                                buf_end = buffer + c;
                        }
                        c = *ptr++;
                        if (c > ' ' && c < 0177) {
                                /* visible ASCII codes */
                                if (!in_word) {
                                        word_count++;
                                        in_word = 1;
                                }
```

```
                        continue;
                }
                if (c == '\n')
                        line_count++;
                else if (c != ' ' && c != '\t')
                        continue;
                in_word = 0;
                /* c is newline, space, or tab */
        }
        wc_print(which, char_count, word_count,
                 line_count);
        if (file_count)
                printf(" %s\n", *argv); /* not stdin */
        else
                printf("\n");   /* stdin */
        close(fd);
        tot_line_count += line_count;
        tot_word_count += word_count;
        tot_char_count += char_count;
        /* even if there is only one file */
} while (--argc > 0);
if (file_count > 1) {
        wc_print(which, tot_char_count,
                 tot_word_count, tot_line_count);
        printf(" total in %d files\n", file_count);
}
exit(status);
```

@

Appendix D

wc.nw"

This is the documented version of the captured literate C program **wc.c.**

```
\makeatletter
\def\idxexample#1{\nwix@id@uses#1}
\makeatother

This program illustrates the incremental redocumentation using literate
programming system.
The original is Ramsey's literate program {\bf wc.nw} which is demonstrated
the use of {\bf noweb}, a tool for literate programming.
The C program {\bf wc.c} is extracted from {\bf wc.nw} by using noweb tool
{\bf notangle}.
The assumption is that the program source code {\bf wc.c} only exists for
redocumentation.
The C program {\bf wc.c} is captured to a literate C program {\bf wc.nwp}
by the capture process.
The captured literate C program {\bf wc.nwp} has cross-reference for chunks
and identifiers and also printed type set documentation by using \LaTeX\.
But this is only a basic achievement of the system.
Here, this illustration shows the incremental redocumentation of the program.
How do we do that?

First, group related includes, definitions and global variables.
Then tie the comments to the source code.
Then tie the comments to the source code.
Move the comment lines to the documentation chunk which exists before the code
chunk.
Now the documentation chunks exists for the appropriate code chunk.
But if there is no comment lines,  a related documentation
chunk for the code chunk cannot be created.
```

Second, use the tools for program comprehension.
The call graph tool, ccg is used as an example for this program.
The call graph is in postscript form and add it to the documentation chunk.
The software engineers can quickly understand the levels of the program and
the interrelationship of the functions.

Similarly, by using various kinds of tools, various kinds of information
could be possibly stored in the documentation chunks, such as,
requirement specifications, design,
data flow, control flow, test data and testing
methods, etc.

Finally, the functionality of the code chunks can be added to the documentation
chunks.
When a software engineer read the program, they can understand
the behaviour of the code chunks by using their experience.
This knowledge can be added to the documentation chunks and these documents
are always tied together with the code chunks.
If the program is retrieved many times by different software engineers
for the purpose of maintenance or reuse, the different views of the
documentation will be added to the documentation chunk.
This documentation cannot be lost and can be seen on the computer screen or
as a typeset document report or a book form for the future.
It will reduce the time for program comprehension in the software
maintenance and reuse phase of the software life cycle.
This is our main goal of incremental redocumentation using literate
programming.

```
<<*>>=
<<Header files to include>>
<<Definitions>>
<<Global variables>>
<<The function [[wc_print]]>>
<<The main program>>
@
<<Header files to include>>=
#include <stdio.h>
@
Group related  definition lines together into one chunk.

{\bf OK}\ is the status code for successful run.\\
{\bf usage_error}\ is the status code for improper syntax.
<<Definitions>>=
#define OK              0
#define usage_error     1
@ %def OK usage_error
```

```
{\bf cannot_open_file}\ is the status code for file access error.\\
{\bf READ_ONLY}\ is the read access code for system open.
<<Definitions>>=
#define cannot_open_file 2
#define READ_ONLY 0
@ %def cannot_open_file READ_ONLY
{\bf buf_size}\ is the stdio.h BUFSIZ chosen for efficiency.
<<Definitions>>=
#define buf_size BUFSIZ
@ %def buf_size
<<Definitions>>=
#define print_count(n) printf("%8ld", n)
@ %def print_count
{\bf status}\ is the exit status of command, initially OK.
<<Global variables>>=
int status = OK;
@ %def status
{\bf prog_name}\ is who we are.
<<Global variables>>=
char *prog_name;
@ %def prog_name
{\bf tot_word_count, tot_line_count, tot_char_count }\ are
the total number of words, lines, and chars.
<<Global variables>>=
long tot_word_count, tot_line_count,
     tot_char_count;
@ %def tot_word_count tot_line_count tot_char_count
{\bf which} is which counts to print.\\
{\bf char_count, word_count, line_count}\ are given totals
<<The function [[wc_print]]>>=
wc_print(which, char_count, word_count, line_count)
  char *which;
  long char_count, word_count, line_count;
{
<<Variables local to [[wc_print]]>>
<<Body of [[wc_print]]>>
}
@ %def wc_print which char_count word_count line_count
<<Variables local to [[wc_print]]>>=

@ %def
<<Body of [[wc_print]]>>=
  while (*which)
    switch (*which++) {
      case 'l': print_count(line_count);
        break;
      case 'w': print_count(word_count);
```

```
        break;
      case 'c': print_count(char_count);
        break;
      default:
        if ((status & usage_error) == 0) {
          fprintf(stderr,
            "\nUsage: %s [-lwc] [filename ...]\n",
            prog_name);
          status |= usage_error;
        }
    }
@
{\bf argc}\ is the number of arguments on UNIX command line.\\
{\bf argv}\ is the arguments, an array of strings.
<<The main program>>=
main(argc, argv)
  int argc;
  char **argv;
{
<<Variables local to [[main]]>>
<<Body of [[main]]>>
}
@
{\bf file_count}\ is how many files there are.\\
{\bf which}\ is which counts to print.
<<Variables local to [[main]]>>=
  int file_count;
  char *which;
@ %def file_count which
{\bf fd} is the file descriptor, initialized to stdin.
<<Variables local to [[main]]>>=
  int fd = 0;
@ %def fd
We read the input into this array {\bf buffer}.\\
{\bf ptr}\ is the first unprocessed character in buffer
{\bf buf_end}\ is the first unused position in buffer
<<Variables local to [[main]]>>=
  char buffer[buf_size];
  register char *ptr;
  register char *buf_end;
@ %def buffer ptr buf_end
{\bf c}\ is the current char, or no. of chars just read.\\
{\bf in_word}\ is are we within a word?
<<Variables local to [[main]]>>=
  register int c;
  int in_word;
@ %def c in_word
```

```
{\bf word_count, line_count, char_count}\ are the
number of words, lines, and chars so far.
<<Variables local to [[main]]>>=
  long word_count, line_count, char_count;
@ %def word_count line_count char_count
<<Body of [[main]]>>=
  prog_name = argv[0];
  which = "lwc";
    /* if no option is given, print 3 values */
  if (argc > 1 && *argv[1] == '-') {
    which = argv[1] + 1;
    argc--;
    argv++;
  }
  file_count = argc - 1;
  argc--;
  do {
    if (file_count > 0
    && (fd = open(*(++argv), READ_ONLY)) < 0) {
      fprintf(stderr,
        "%s: cannot open file %s\n",
        prog_name, *argv);
      status |= cannot_open_file;
      file_count--;
      continue;
    }
    ptr = buf_end = buffer;
    line_count = word_count = char_count = 0;
    in_word = 0;
    while (1) {
      if (ptr >= buf_end) {
        ptr = buffer;
        c = read(fd, ptr, buf_size);
        if (c <= 0) break;
        char_count += c;
        buf_end = buffer + c;
      }
      c = *ptr++;
      if (c > ' ' && c < 0177) {
        /* visible ASCII codes */
        if (!in_word) {
          word_count++;
          in_word = 1;
        }
        continue;
      }
      if (c == '\n') line_count++;
```

```
      else if (c != ' ' && c != '\t') continue;
      in_word = 0;
        /* c is newline, space, or tab */
    }
    wc_print(which, char_count, word_count,
             line_count);
    if (file_count)
      printf(" %s\n", *argv); /* not stdin */
    else
      printf("\n");           /* stdin */
    close(fd);
    tot_line_count += line_count;
    tot_word_count += word_count;
    tot_char_count += char_count;
      /* even if there is only one file */
  } while (--argc > 0);
  if (file_count > 1) {
    wc_print(which, tot_char_count,
             tot_word_count, tot_line_count);
    printf(" total in %d files\n", file_count);
  }
  exit(status);
@
\section{List of code chunks}
This list is generated automatically.
The numeral is that of the first definition of the chunk.
\nowebchunks
\section{Index}
Here is a list of the identifiers used, and where they appear.
Underlined entries indicate the place of definition.
This index is generated automatically.
\nowebindex
```

Appendix E

lines.c

```
#include <stdio.h>
#include <string.h>
#define MAXLINES 10 /* max #lines to be sorted */
#define MAXLEN   30 /* length of input line */
#define ALLOCSIZE 100 /* available space */
static char allocbuf[ALLOCSIZE];
static char *allocp = allocbuf;
char *lineptr[MAXLINES];
char *alloc(n)
int n;
{

 if (allocbuf + ALLOCSIZE - allocp >= n)
   {
    allocp += n;
    return allocp - n;
   }
 else
   return 0;
}
int getline (s, lim)
 char s[];
 int lim;
{
 int c,i;
 i = 0;

 while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
       s[i++] = c;
 if (c == '\n')
       s[i++] = c;
 s[i] = '\0';
```

```
 return i;
}
int readlines(lineptr, maxlines)
char *lineptr[];
int maxlines;
{
 int len, nlines;
 char *p, line[MAXLEN];
 nlines = 0;
 while ((len = getline(line, MAXLEN)) > 0)
       {
       if (nlines >= maxlines)
         return -1;
       if ((p =alloc(len)) == NULL)
          return -1;
       line[len-1] = '\0';
       strcpy(p,line);
       lineptr[nlines++] = p;
       }
 return nlines;
}
writelines( lineptr, nlines)
char *lineptr[];
int nlines;
{

 while (nlines-- > 0)
       printf("%s\n", *lineptr++);
}
swap(v, i, j)
char *v[];
int i,j;
{
 char *temp;
 temp = v[i];
 v[i] = v[j];
 v[j] = temp;
}
qsort(v, left, right)
char *v[];
int left, right;
{
 int i, last;
 if(left >= right)
   return;
 swap(v, left, (left+right)/2);
 last = left;
```

```
 for ( i=left+1; i <= right; i++)
     if (strcmp(v[i], v[left]) < 0)
        swap (v, ++last, i);
 swap(v, left, last);
 qsort(v, left, last-1);
 qsort(v, last+1, right);
}
main()
{
 last = left;
 for ( i=left+1; i <= right; i++)
     if (strcmp(v[i], v[left]) < 0)
        swap (v, ++last, i);
 swap(v, left, last);
 qsort(v, left, last-1);
 qsort(v, last+1, right);
}
main()
{
int nlines;
 if ((nlines = readlines(lineptr, MAXLINES)) >= 0)
    {
     qsort(lineptr,0, nlines-1);
     writelines(lineptr, nlines);
     return 0;
    }
 else
    {
     printf("error : input too big to sort\m");
     return 1;
    }
}
 last = left;
 for ( i=left+1; i <= right; i++)
     if (strcmp(v[i], v[left]) < 0)
        swap (v, ++last, i);
 swap(v, left, last);
 qsort(v, left, last-1);
 qsort(v, last+1, right);
}
main()
{
int nlines;
 if ((nlines = readlines(lineptr, MAXLINES)) >= 0)
    {
     qsort(lineptr,0, nlines-1);
     writelines(lineptr, nlines);
```

```
     return 0;
    }
 else
    {
     printf("error : input too big to sort\m");
     return 1;
    }
}
```

Appendix F

lines.nw'

```
\makeatletter
\def\idxexample#1{\nwix@id@uses#1}
\makeatother

<<*>>=
<<Headerfiles to include>>
<<Definitions>>
<<Global Variables>>
<<The function [[alloc]]>>
<<The function [[getline]]>>
<<The function [[readlines]]>>
<<The function [[writelines]]>>
<<The function [[swap]]>>
<<The function [[qsort]]>>
<<The main program>>
@
<<Headerfiles to include>>=
#include <stdio.h>
@
<<Headerfiles to include>>=
#include <string.h>
@
<<Definitions>>=
#define MAXLINES 10 /* max #lines to be sorted */S
@ %def MAXLINES
<<Definitions>>=
#define MAXLEN   30 /* length of input line */
@ %def MAXLEN
<<Definitions>>=
#define ALLOCSIZE 100 /* available space */
@ %def ALLOCSIZE
<<Global Variables>>=
```

```
static char allocbuf[ALLOCSIZE];
@ %def allocbuf ALLOCSIZE
<<Global Variables>>=
static char *allocp = allocbuf;
@ %def allocp allocbuf
<<Global Variables>>=
char *lineptr[MAXLINES];
@ %def lineptr MAXLINES
<<The function [[alloc]]>>=
char *alloc(n)
int n;
{
<<Variables local to [[alloc]]>>
<<Body of [[alloc]]>>
}
@ %def alloc n

<<Variables local to [[alloc]]>>=

@ %def

<<Body of [[alloc]]>>=
 if (allocbuf + ALLOCSIZE - allocp >= n)
   {
    allocp += n;
    return allocp - n;
   }
 else
   return 0;
@
<<The function [[getline]]>>=
int getline (s, lim)
 char s[];
 int lim;
{
<<Variables local to [[getline]]>>
<<Body of [[getline]]>>
}
@ %def getline s lim
<<Variables local to [[getline]]>>=
 int c,i;
@ %def c i
<<Body of [[getline]]>>=
 i = 0;

 while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
       s[i++] = c;
```

```
 if (c == '\n')
       s[i++] = c;
 s[i] = '\0';
 return i;
@
<<The function [[readlines]]>>=
int readlines(lineptr, maxlines)
char *lineptr[];
int maxlines;
{
<<Variables local to [[readlines]]>>
<<Body of [[readlines]]>>
}
@ %def readlines lineptr maxlines
<<Variables local to [[readlines]]>>=
 int len, nlines;
@ %def len nlines
<<Variables local to [[readlines]]>>=
 char *p, line[MAXLEN];
@ %def p line
<<Body of [[readlines]]>>=
 nlines = 0;
 while ((len = getline(line, MAXLEN)) > 0)
       {
       if (nlines >= maxlines)
          return -1;
       if ((p =alloc(len)) == NULL)
          return -1;
       line[len-1] = '\0';
       strcpy(p,line);
       lineptr[nlines++] = p;
       }
 return nlines;
@
<<The function [[writelines]]>>=
writelines( lineptr, nlines)
char *lineptr[];
int nlines;
{
<<Variables local to [[writelines]]>>
<<Body of [[writelines]]>>
}
@ %def writelines lineptr nlines
<<Variables local to [[writelines]]>>=

@ %def
<<Body of [[writelines]]>>=
```

```
 while (nlines-- > 0)
       printf("%s\n", *lineptr++);
@
<<The function [[swap]]>>=
swap(v, i, j)
char *v[];
int i,j;
{
<<Variables local to [[swap]]>>
<<Body of [[swap]]>>
}
@ %def swap v i j
<<Variables local to [[swap]]>>=
 char *temp;
@ %def temp
<<Body of [[swap]]>>=
 temp = v[i];
 v[i] = v[j];
 v[j] = temp;
@
<<The function [[qsort]]>>=
qsort(v, left, right)
char *v[];
int left, right;
{
<<Variables local to [[qsort]]>>
<<Body of [[qsort]]>>
}
@ %def qsort v left right
<<Variables local to [[qsort]]>>=
 int i, last;
@ %def i last
<<Body of [[qsort]]>>=
 if(left >= right)
   return;
 swap(v, left, (left+right)/2);
 last = left;
 for ( i=left+1; i <= right; i++)
     if (strcmp(v[i], v[left]) < 0)
        swap (v, ++last, i);
 swap(v, left, last);
 qsort(v, left, last-1);
 qsort(v, last+1, right);
@
<<The main program>>=
main()
{
```

```
<<Variables local to [[main]]>>
<<Body of [[main]]>>
}
@ %def main
<<Variables local to [[main]]>>=
int nlines;
@ %def nlines
<<Body of [[main]]>>=
 if ((nlines = readlines(lineptr, MAXLINES)) >= 0)
    {
     qsort(lineptr,0, nlines-1);
     writelines(lineptr, nlines);
     return 0;
    }
 else
    {
     printf("error : input too big to sort\m");
     return 1;
    }
@
\section{List of code chunks}
This list is generated automatically.
The numeral is that of the first definition of the chunk.
\nowebchunks
\section{Index}
Here is a list of the identifiers used, and where they appear.
Underlined entries indicate the place of definition.
This index is generated automatically.
\nowebindex
```

Appendix G

lines.nw"

```
\makeatletter
\def\idxexample#1{\nwix@id@uses#1}
\makeatother

<<*>>=
<<Headerfiles to include>>
<<Definitions>>
<<Global Variables>>
<<The function [[alloc]]>>
<<The function [[getline]]>>
<<The function [[readlines]]>>
<<The function [[writelines]]>>
<<The function [[swap]]>>
<<The function [[qsort]]>>
<<The main program>>
@
<<Headerfiles to include>>=
#include <stdio.h>
@
<<Headerfiles to include>>=
#include <string.h>
@
<<Definitions>>=
#define MAXLINES 10 /* max #lines to be sorted */S
@ %def MAXLINES
<<Definitions>>=
#define MAXLEN   30 /* length of input line */
@ %def MAXLEN
<<Definitions>>=
#define ALLOCSIZE 100 /* available space */
@ %def ALLOCSIZE
<<Global Variables>>=
```

```
static char allocbuf[ALLOCSIZE];
@ %def allocbuf ALLOCSIZE
<<Global Variables>>=
static char *allocp = allocbuf;
@ %def allocp allocbuf
<<Global Variables>>=
char *lineptr[MAXLINES];
@ %def lineptr MAXLINES
<<The function [[alloc]]>>=
char *alloc(n)
int n;
{
<<Variables local to [[alloc]]>>
<<Body of [[alloc]]>>
}
@ %def alloc n

<<Variables local to [[alloc]]>>=

@ %def

<<Body of [[alloc]]>>=
 if (allocbuf + ALLOCSIZE - allocp >= n)
   {
    allocp += n;
    return allocp - n;
   }
 else
   return 0;
@
<<The function [[getline]]>>=
int getline (s, lim)
 char s[];
 int lim;
{
<<Variables local to [[getline]]>>
<<Body of [[getline]]>>
}
@ %def getline s lim
<<Variables local to [[getline]]>>=
 int c,i;
@ %def c i
<<Body of [[getline]]>>=
 i = 0;

 while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
       s[i++] = c;
```

```
 if (c == '\n')
       s[i++] = c;
 s[i] = '\0';
 return i;
@
<<The function [[readlines]]>>=
int readlines(lineptr, maxlines)
char *lineptr[];
int maxlines;
{
<<Variables local to [[readlines]]>>
<<Body of [[readlines]]>>
}
@ %def readlines lineptr maxlines
<<Variables local to [[readlines]]>>=
 int len, nlines;
@ %def len nlines
<<Variables local to [[readlines]]>>=
 char *p, line[MAXLEN];
@ %def p line
<<Body of [[readlines]]>>=
 nlines = 0;
 while ((len = getline(line, MAXLEN)) > 0)
       {
       if (nlines >= maxlines)
          return -1;
       if ((p =alloc(len)) == NULL)
          return -1;
       line[len-1] = '\0';
       strcpy(p,line);
       lineptr[nlines++] = p;
       }
 return nlines;
@
<<The function [[writelines]]>>=
writelines( lineptr, nlines)
char *lineptr[];
int nlines;
{
<<Variables local to [[writelines]]>>
<<Body of [[writelines]]>>
}
@ %def writelines lineptr nlines
<<Variables local to [[writelines]]>>=

@ %def
<<Body of [[writelines]]>>=
```

```
 while (nlines-- > 0)
       printf("%s\n", *lineptr++);
@
<<The function [[swap]]>>=
swap(v, i, j)
char *v[];
int i,j;
{
<<Variables local to [[swap]]>>
<<Body of [[swap]]>>
}
@ %def swap v i j
<<Variables local to [[swap]]>>=
 char *temp;
@ %def temp
<<Body of [[swap]]>>=
 temp = v[i];
 v[i] = v[j];
 v[j] = temp;
@
<<The function [[qsort]]>>=
qsort(v, left, right)
char *v[];
int left, right;
{
<<Variables local to [[qsort]]>>
<<Body of [[qsort]]>>
}
@ %def qsort v left right
<<Variables local to [[qsort]]>>=
 int i, last;
@ %def i last
<<Body of [[qsort]]>>=
 if(left >= right)
   return;
 swap(v, left, (left+right)/2);
 last = left;
 for ( i=left+1; i <= right; i++)
     if (strcmp(v[i], v[left]) < 0)
        swap (v, ++last, i);
 swap(v, left, last);
 qsort(v, left, last-1);
 qsort(v, last+1, right);
@
<<The main program>>=
main()
{
```

```
<<Variables local to [[main]]>>
<<Body of [[main]]>>
}
@ %def main
<<Variables local to [[main]]>>=
int nlines;
@ %def nlines
<<Body of [[main]]>>=
 if ((nlines = readlines(lineptr, MAXLINES)) >= 0)
    {
     qsort(lineptr,0, nlines-1);
     writelines(lineptr, nlines);
     return 0;
    }
 else
    {
     printf("error : input too big to sort\m");
     return 1;
    }
@
\section{List of code chunks}
This list is generated automatically.
The numeral is that of the first definition of the chunk.
\nowebchunks
\section{Index}
Here is a list of the identifiers used, and where they appear.
Underlined entries indicate the place of definition.
This index is generated automatically.
\nowebindex
```

Appendix H

gen-nw

```
\makeatletter
\def\idxexample#1{\nwix@id@uses#1}
\makeatother

<<*>>=
<<Headerfiles to include>>
<<Definitions>>
<<The function [[process_file]]>>
<<The function [[process_file_to_end]]>>
<<The main program>>
@
<<Headerfiles to include>>=
#include <stdio.h>
@
<<Headerfiles to include>>=
#include <string.h>
@
<<Definitions>>=
#define MAXSTRING 127
@ %def  MAXSTRING
void
<<The function[[process_file]]>>=
process_file(outfd, infd, infilename, fromline, toline, fname, ptype)
   FILE *outfd, *infd;
   char *infilename;
   int *fromline;
   int toline;
   char *fname;
   char ptype;
{
<<Variables local to [[process_file]]>>
```

```
<<Body of [[process_file]]>>
}
@ %def process_file outfd  infd  infilename  fromline  toline  fname  ptype

<<Variables local to [[process_file]]>>=
   int currline;
@ %def currline
<<Variables local to [[process_file]]>>=
   int varlength;
@ %def varlength
<<Variables local to [[process_file]]>>=
   char inline[MAXSTRING];
@ %def inline
<<Variables local to [[process_file]]>>=
   char defvar[MAXSTRING];
@ %def defvar
<<Variables local to [[process_file]]>>=
   char outline[MAXSTRING];
@ %def outline
<<Variables local to [[process_file]]>>=
   char functionheader[MAXSTRING];
@ %def functionheader
<<Variables local to [[process_file]]>>=
   char *functionnamestart, *tmpstr;
@ %def functionnamestart
<<Variables local to [[process_file]]>>=
   char *varstart;
@ %def varstart
<<Body of [[process_file]]>>=
   currline = *fromline;
   while (currline != toline) {
      if (fgets(inline, MAXSTRING, infd) == NULL) {
         fprintf(stderr, "Unexpected EOF at line %d in reading file %s from
%d to %d\n",
                  currline, infilename, *fromline, toline);
         exit(1);
      }
      if (strncmp(inline, "#include", 8) == 0) {
         fprintf(outfd, "<<Headerfiles to include>>=\n");
         printf("<<Headerfiles to include>>=\n");
         fprintf(outfd, "%s", inline);
         printf("%s", inline);
         fprintf(outfd, "@\n");
         printf("@\n");
      } else {
         fprintf(outfd, "%s", inline);
      }
```

```
      currline++;
   }
   fgets(inline, MAXSTRING, infd);
   if (ptype == 'T') {
      if (strncmp(inline, "main", 4) == 0) {
         fprintf(outfd, "<<The main program>>=\n");
         fprintf(outfd, "%s", inline);
         printf("<<The main program>>=\n");
         printf("%s", inline);
      } else {
         fprintf(outfd, "<<Functions>>=\n");
         fprintf(outfd, "%s", inline);
         printf("<<Functions>>=\n");
         printf("%s", inline);
      }


      fprintf(outfd, "@ %%def %s\n", fname);
      printf("@ %%def %s\n", fname);
      *fromline += 1;

   } else {
      if (strncmp(inline, "#define", 7) == 0) {
         printf("<<Definitions>>=\n");
         fprintf(outfd, "<<Definitions>>=\n");
         outline[0] = 0;
         printf("%s", inline);
         fprintf(outfd, "%s", inline);
         varstart = strstr(inline, " ");
         varlength = strcspn(varstart + 1, " ");
         (void) strncpy(defvar, varstart, varlength + 1);
         defvar[varlength + 1] = 0;
         (void) strcat(outline, defvar);
         printf("@ %%def %s\n", outline);
         fprintf(outfd, "@ %%def %s\n", outline);
      } else {
         printf("<<Global Variables>>=\n");
         fprintf(outfd, "<<Global Variables>>=\n");
         fprintf(outfd, "%s", inline);
         fprintf(outfd, "@ %%def %s\n", fname);
         printf("%s\n", inline);
         printf("@ %%def %s\n", fname);
      }

   }
   *fromline = toline + 1;
}
```


```
@

<<The function[[process_file_to_end]]>>=
void process_file_to_end(outfd, infd, fname)
   FILE *outfd, *infd;
   char *fname;
{
<<Variables local to [[process_file_to_end]]>>
<<Body of [[process_file_to_end]]>>
}
@ %def process_file_to_end outfd  infd  fname

<<Variables local to [[process_file_to_end]]>>=
   char inline[MAXSTRING];
@ %def inline
<<Variables local to [[process_file_to_end]]>>=
   char outline[MAXSTRING];
@ %def outline
<<Variables local to [[process_file_to_end]]>>=
   char defvar[MAXSTRING];
@ %def defvar
<<Variables local to [[process_file_to_end]]>>=
   int varlength;
@ %def varlength
<<Variables local to [[process_file_to_end]]>>=
   char *varstart;
@ %def varstart
<<Body of [[process_file_to_end]]>>=
   while (fgets(inline, MAXSTRING, infd) != NULL) {
      if (strncmp(inline, "#define", 7) == 0) {
         fprintf(outfd, "<<Definitions>>=\n");
         outline[0] = 0;
         fprintf(outfd, "%s", inline);
         varstart = strstr(inline, " ");
         varlength = strcspn(varstart + 1, " ");
         (void) strncpy(defvar, varstart, varlength + 1);
         defvar[varlength + 1] = 0;
         (void) strcat(outline, defvar);
         fprintf(outfd, "@ %%def %s\n", outline);
      } else {
         fprintf(outfd, "%s", inline);

      /*
       * printf("Process file to end function"); printf("Copying line |%s",
       * inline);
       */
      }
```

```
   }
}
@
void
<<The main program>>=
main(argc, argv)
   int argc;
   char **argv;
{
<<Variables local to [[main]]>>
<<Body of [[main]]>>
}
@ %def main argc  argv

<<Variables local to [[main]]>>=
   FILE *infp, *outfp;
@ %def infp
<<Variables local to [[main]]>>=
   FILE *refedfp;
@ %def refedfp
<<Variables local to [[main]]>>=
   char *infile, *outfile;
@ %def infile
<<Variables local to [[main]]>>=
   char functionname[MAXSTRING], filename[MAXSTRING];
@ %def filename
<<Variables local to [[main]]>>=
   int linenumber, count;
@ %def count
<<Variables local to [[main]]>>=
   char refed_filename[MAXSTRING];
@ %def refed_filename
<<Variables local to [[main]]>>=
   int refed_fileOK = 0;
@ %def refed_fileOK
<<Variables local to [[main]]>>=
   int refed_currentline = 1;
@ %def refed_currentline
<<Variables local to [[main]]>>=
   char c, chstring[2], type;
@ %def c
<<Variables local to [[main]]>>=
   int tline = 0;
@ %def tline
<<Variables local to [[([main]]>>=
   int xline = 0;
@ %def xline
```

```
<<Body of [[main]]>>=
   if (argc != 3) {
      fprintf(stderr, "\nUsage gen-nw infile outfile\n");
      exit;
   }

/*
 * printf("Input file name: "); scanf("%s", infile); printf("Output file
 * name: "); scanf("%s", outfile);
 *
 */
   infile = argv[1];
   outfile = argv[2];
   if ((outfp = fopen(outfile, "w")) != NULL) {
      if ((infp = fopen(infile, "r")) != NULL) {
         printf("Information from Info file '%s'\n\n", infile);
         refed_filename[0] = '\0';
         refed_fileOK = 0;
         fprintf(outfp, "<<*>>\n");
         fprintf(outfp, "<<Headerfiles to include>>\n");
         fprintf(outfp, "<<Definitions>>\n");
         fprintf(outfp, "<<Global Variables>>\n");
         fprintf(outfp, "<<Functions>>\n");
         fprintf(outfp, "<<The main program>>\n");
         fprintf(outfp, "@\n");
         while (fscanf(infp, "%1c%s%s%d\n", &type, filename, functionname,
&linenumber) != EOF) {
            printf("Type = '%c', Filename = '%s', Function Name ='%s',
Linenumber = %d\n ", type, filename, functionname, linenumber);
            chstring[0] = ' ';
            chstring[1] = '\0';
            if (strcmp(filename, refed_filename) != 0) {
               if (refed_filename[0] != '\0')
                  process_file_to_end(outfp, refedfp, functionname);
               (void) strcpy(refed_filename, filename);
               if ((refedfp = fopen(refed_filename, "r")) != NULL) {
                  refed_fileOK = 1;
                  refed_currentline = 1;
               } else {
                  printf("Can't open input file '%s'\n", refed_filename);
                  refed_fileOK = 0;
               }
            }
            if (refed_fileOK == 1) {
            /* now process the file */
               printf("Reading line %d from file '%s'\n", linenumber,
refed_filename);
```

```
                if (type == 'T') {
                   tline = linenumber;
                   process_file(outfp, refedfp, refed_filename,
&refed_currentline, linenumber, functionname, type);
                } else {
                   if (type == 'X') {
                      if (linenumber != tline && linenumber != xline) {
                         process_file(outfp, refedfp, refed_filename,
&refed_currentline, linenumber, functionname, type);
                         xline = linenumber;
                      } else {
                         if (linenumber == tline) {
                            tline = 0;
                            xline = linenumber;
                         } else {
                            if (linenumber == xline) {

         fprintf(outfp, "<<Functions>>\n");
         fprintf(outfp, "<<The main program>>\n");
         fprintf(outfp, "@\n");
         while (fscanf(infp, "%1c%s%s%d\n", &type, filename, functionname,
&linenumber) != EOF) {
            printf("Type = '%c', Filename = '%s', Function Name ='%s',
Linenumber = %d\n ", type, filename, functionname, linenumber);
            chstring[0] = ' ';
            chstring[1] = '\0';
            if (strcmp(filename, refed_filename) != 0) {
               if (refed_filename[0] != '\0')
                  process_file_to_end(outfp, refedfp, functionname);
               (void) strcpy(refed_filename, filename);
               if ((refedfp = fopen(refed_filename, "r")) != NULL) {
                  refed_fileOK = 1;
                  refed_currentline = 1;
               } else {
                  printf("Can't open input file '%s'\n", refed_filename);
                  refed_fileOK = 0;
               }
            }
            if (refed_fileOK == 1) {
            /* now process the file */
               printf("Reading line %d from file '%s'\n", linenumber,
refed_filename);
               if (type == 'T') {
                  tline = linenumber;
                  process_file(outfp, refedfp, refed_filename,
&refed_currentline, linenumber, functionname, type);
               } else {
```

```
                    if (type == 'X') {
                       if (linenumber != tline && linenumber != xline) {
                          process_file(outfp, refedfp, refed_filename,
&refed_currentline, linenumber, functionname, type);
                          xline = linenumber;
                       } else {
                          if (linenumber == tline) {
                             tline = 0;
                             xline = linenumber;
                          } else {
                             if (linenumber == xline) {
                                fprintf(outfp, "@ %%def %s\n", functionname);
                                printf("@ %%def %s\n", functionname);
                             }
                          }
                       }
                    }
                 }
              }
           }
           process_file_to_end(outfp, refedfp, filename);
        } else {
           printf("Can't open input file '%s'\n", infile);
        }
     } else {
        printf("Can't open output file '%s'\n", outfile);
     }

}
@
```

Bibliography

[1] Avenarius, A. and Opperman, S., A Literate Programming System for FORTRAN8X, *ACM SIGPLAN notices, Vol. 25, No. 1*, 1990 pp. 52-58

[2] Bennett, K. H., Legacy Systems: Coping with Success, *IEEE Software*, 1995 pp. 19-23

[3] Bennett K., Cornelius B., Munro M. and Robson D., Software Maintenance, *in McDermid, J.A. Software Engineer's Handbook, Chp 20, Butterworth Heinemann*, 1991

[4] Bently, J., Programming Pearls - Literate Programming, *Communication of the ACM, Vol. 29, No. 5*, 1986 pp. 364-369

[5] Bently, J., Knuth, D. E. and McIlroy, D., Programming Pearls - Literate Programming, *Communication of the ACM, Vol. 29, No. 6*, 1986 pp.471-483

[6] Bently, J., and Gries D., Programming Pearls - Literate Programming, *Communication of the ACM, Vol. 30, No. 4*, 1987 pp. 284-290

[7] Bishop, J. M. and Gregson, K. M., Literate Programming and the (LIPED) Environment, *Journal of Structured Programming, Vol. 13, No. 1*, 1992 pp. 23-34

[8] Proceedings of the Workshop on Program Comprehension, Greater Understanding through Maintainer Driven Traceability, *IEEE Press*, 1996 pp. 100-106

[9] Bottaci, L. and Steward, A., Extending Software into the Future, *Hypermedia/Hypertext and Object-oriented Databases, UNICOM*, 1991 pp. 219-235

[10] Brown, M. E. and Childs, B., An Interactive Environment for Literate Programming, *Journal of Structured Programming, Vol. 11, No. 1*, 1990 pp. 11-25

[11] Brown, M. E. and Cordes, D., Literate Programming Applied to Conventional Software Design, *Journal of Structured Programming, Vol. 11, No. 1*, 1990 pp. 85-98

[12] Brown, M. E. and Czejdo, B., A Hypertext for Literate Programming, *Advances in Computing and Information–ICCI '90/*, 1990 pp. 85-98

[13] Chapin, N., Software Maintenance: a different view, *AFIPS Conference Proceedings, 54th National Computer Conference*, 1985 pp. 509-513

[14] Conklin, J., Hypertext: An introduction and survey, *IEEE Computer, Vol. 20*, pp. 17-40

[15] Denning, P. J., Announcing Literate Programming, *Communications of the ACM, Vol. 30, No. 7*, 1987 pp. 593

[16] Fletton, N. T. and Munro, M., Redocumenting Software Systems using Hypertext Technology, *Conference on Software Maintenance, IEEE*, 1988 pp. 54-59

[17] Fletton, N.T., Documenting for Software Maintenance: the redocumentation of existing systems, *MSc. Thesis, University of Durham*, 1988

[18] Freeman, R.M., Software Maintenance: Redocumentation of Existing COBOL Systems using Hypertext Technology, *MSc. Thesis, University of Durham*, 1992

[19] Gries, D., Reply to Knuth's Small Work of Literature. In Programming Pearls, *Communications of the ACM, Vol. 30, No. 4*, 1987 pp. 286-288

[20] Guillemetter, R.A., Application Software Documentation: a reader's measure, *PhD Thesis, University of Houston*, May, 1986

[21] Hamilton, E., Literate Programming – Expanding Generalized Regular Expressions, *Communications of the ACM, Vol. 31, No. 12*, 1988 pp. 1376-1385

[22] Hanson, D. R., Literate programming – Printing Common Words, *Communications of the ACM, Vol. 30, No. 7*, 1987 pp. 594-599

[23] Hyman, M. S., Literate C++, *Computer Language, Vol. 7, No. 7*, 1990 pp. 67-68, 70-72, 74-77, 79

[24] Jackson, M. A., Literate Programming - Processing Transactions, *Communications of the ACM, Vol. 30, No. 12*, 1987 pp. 1000-1010

[25] Kernighan, B.W. and Ritchie D.M., The C Programming Language, *Prentice-Hall*, 1978

[26] Knuth, D. E., The WEB System of Structured Documentation, *Stanford Computer Science Report, CS 980*, 1983

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

[28] Knuth, D. E., A Small Work of Literature, In Programming Pearls, *Communications of the ACM, Vol. 21, No. 5*, 1986 pp. 366-367

[29] Knuth, D. E., Mini-indexes for literate programs, *Software-Concepts and Tools, Vol. 15, No. 1*, 1994 pp. 2-11

[30] Kortright, E. and Cordes, D., Cnest and Cscope: Tools for the literate programming environment, *Proceedings / IEEE Southeastcon '92, Vol. 2*, 1992 pp. 604-609

[31] Levy, S., WEB adapted to C, another approach, *Journal of TUGboat, Vol. 8, No. 1*, 1987 pp. 12-13

[32] Levy, S., Literate Programming and CWEB, *Computer Language, Vol. 10, No. 1*, 1993 pp.67-68, 70

[33] Lientz, B.P. and Swanson, E. B., Software Maintenance Management, *Addison Wesley*, 1980

[34] Lindsay, D. C., Literate Programming-A File Difference Program, *Communications of the ACM, Vol. 32, No. 6*, 1989 pp. 740-755

[35] Osterbye, K. and Normark, K., An interaction engine for rich hypertexts, *ECHT'94 Proc. ACM European Conf. Hypertext*, 1994 pp. 167-176

[36] Osterbye, K., Literate Smalltalk Programming Using Hypertext, *IEEE Transaction on Software Engineering, Vol. 31, No. 2*, 1995 pp. 138-145

[37] Parnas, D. L., Software Aging, *Proceedings: 16th International Conference on Software Engineering*, 1994 pp. 279-287

[38] Ramsdell, J. D., Simple support for literate programming in Lisp, *Journal of Tex-Hax. Vol. 88, No. 39*, 1988

[39] Ramsey, N., Weaving a Language-Independent WEB, *Communications of the ACM, Vol. 32, No. 9*, 1989 pp. 1051-1055

[40] Ramsey, N., Literate-Programming Tools Need Not Be Complex, *Technical Report, Princeton University*, 1992

[41] Ramsey, N., Literate Programming Simplified, *Journal of IEEE Software, Vol. 11, No. 5*, 1994 pp. 97-105

[42] Robson, D.J., Bennett, K.H., Cornelius, B.J., and Munro, M., Approaches to Program Comprehension, *Journal of Systems Software, Vol. 14*, 1991 pp. 79-84

[43] Sewell, E.W., How to MANGLE Your Software: the WEB System for Modula-2, *Journal of TUGboat, Vol. 8, No. 2*, 1987 pp.118-122

[44] Standish, T.a., An Essay on Software Reuse, *Transactions on Software Engineering, Vol. 10, No. 5*, 1984 pp. 494-497

[45] Thimbleby, H. W., Literate Programming in C, *Technical Report, University of York*, 1984

[46] Thimbleby, H. W., Experiences of Literate Programming using CWEB, *The Computer Journal, Vol. 29, No. 3*, 1986 pp. 201-211

[47] Van Wyk, C. J., Literate Programming - An Assessment, *Communications of the ACM, Vol. 31, No. 7*, 1990 pp. 343-344

[48] Wong, K., Tilley, S.,Muller, H. and Storey, M., Structural Redocumentation: A Case Study, *IEEE Software, Jan*, 1995 pp. 46-54

[49] Younger, E., Documentation, Chapter 8, *The REDO Compendium*, 1993 pp. 111-121

[50] van Zuylen, H. J., Introduction, *The REDO Compendium*, 1993 pp. 1-9

[51] van Zuylen, H. J., Understanding Reverse Engineering, Chapter 6, *The REDO Compendium*, 1993 pp. 81-92