Structured Editing of Literate Programs

Angus John Charles Duggan

M.Phil.

University of Edinburgh

1994

Abstract

This thesis describes an investigation into the use of syntax-directed editing for programs developed within the literate programming software methodology.

A review of literate programming literature and a discussion of some of the problems which have prevented the widespread acceptance of literate programming is presented. The way in which syntax-directed editing could be used to solve these problems and fulfil four basic criteria for literate programming systems was investigated.

Two implementation studies in creating syntax directed literate program editors were performed, using the Synthesizer Generator, a system for building syntax-directed editors. The compromises made in the design and implementation of these editors with the Synthesizer Generator are discussed. A larger-scale implementation of a literate program editor for the Pascal programming language was performed, using the techniques developed in the implementation studies. The implementation of the Pascal editor highlighted a fundamental problem in the evaluation of the attribute grammar representation of the program which was used.

The problem in the implementation of the Pascal editor was investigated, and it is shown that the problem will necessarily occur when attribute grammars are used to represent literate programs using the techniques developed. It is also shown that the problem is solvable using a modified algorithm for evaluating the attribute grammar, but the running time of this algorithm is unacceptably high for use in interactive systems.

A method for describing the structure of literate programs is proposed, and the potential problems of implementing syntax-directed editors based on this method are examined. A generalisation of this descriptive method is presented which may be applicable to a wide class of non-linear documents.

Acknowledgements

I would like to thank Jenny, for patience, comfort, and support while writing this thesis. I would also like to thank Mandy, for constant optimism and exemplifying perseverance. Finally, I would like to thank Nick and Chris, for making dull times interesting.

Part of this work was funded by an industrial scholarship from STC.

Declaration

I declare that this thesis was composed by myself, and that the work contained in it is my own, except where explicitly stated in the text.

Table of Contents

List of Figures

1 Introduction

1.1 Overview

This thesis describes an investigation into the uses of syntax-directed editing for programs developed with the literate programming software methodology.

Chapter 1 describes the structure of the thesis, presents a brief introduction to literate programming and syntax-directed editing, and explains the motivation for applying syntax-directed editing to literate programs.

Chapter 2 presents an extensive literature survey of literate programming and a set of four basic criteria by which a programming system may be judged literate. A short survey of syntax-directed editing systems is also be presented.

Chapter 3 discusses in more detail the problems which have prevented the widespread acceptance of literate programming, and the way in which syntax-directed editing could be used to solve these problems. The use of syntax-directed editing in relation to the four basic criteria for literate programming systems presented in chapter 2 is discussed, and a strategy for building a syntax-directed literate program editor to demonstrate these points is shown.

Chapter 4 discusses several implementations of literate program editors which were developed using a system for building syntax-directed editors. Two initial feasibility studies which were performed are described, and conclusions about the design and implementation of literate program editors are drawn from them. The implementation of a literate program editor for a real programming language is then described, and the problems encountered in scaling up the techniques developed in the feasibility studies are discussed.

Chapter 5 presents an evaluation of the literate programming editor implemented in chapter 4 with reference to the criteria for literate programming systems given in chapter 3. The problem which prevented the implementation from being viable is examined in more detail, and it is shown that the problem is due to the representation of the literate programs used in chapter 4. A solution to the problem is then shown, but an analysis of its running time reveals that the representation used was inappropriate for literate programs.

Chapter 6 proposes a modification of the representation used for literate programs in chapter 4. The potential problems of implementing syntax-directed editors using this representation are examined, and a generalisation of this new representation is presented which may be applicable to other non-linear documents.

Chapter 7 summarises the results achieved, draws some conclusions about the representation of literate programs in syntax-directed editing and other programming environments, and highlights the work which remains to be done to make syntax-directed editing of literate programs viable.

In this thesis, I assert that the attribute grammar model used by many syntax-directed editors is not sufficient for manipulating literate programs, but that a modification of it may be sufficient for representing not only literate programs, but a variety of other non-linear structures too.

1.2 What is Literate Programming?

The literate programming software methodology was created to address the problem of keeping up-to-date documentation for programs. Brooks ([19], p. 121) asserts that software maintenance is typically 40% or more of the cost of program development. Oman and Cook [85] state that "programmers spend between 47 and 62% of their time trying to comprehend code". It is obvious from these assertions that good documentation is needed to be able to understand and maintain programs.

The term *literate programming* was coined by Donald Knuth [66] to explain the nature of the new software methodology—that by considering programs as works of literature the structure of the program and the accompanying documentation can be improved. Knuth also admits to some malice in naming the methodology; after being forced to adopt structured programming because he did not want to be accused of *unstructured* programming, Knuth got his own back by making other people consider whether they are writing *illiterate* programs.

The essence of the methodology is contained in the idea that instead of writing a program with the objective of instructing a *computer* what to do, we should write the program with the objective of explaining to *human beings* what we want the computer to do. Literate programming can be seen as technical writing applied to programs, where the programmer states everything twice, once informally (the documentation), and once formally (the program). Literate programming allows the programmer to combine the best features of top-down and bottom-up program design methods. Top-down design imparts a strong sense of direction to programming, but keeps a lot of details in suspense until the end. Bottom-up design continually improves the toolkit with which the program is being built, but postpones the overall program organisation to the last minute. With literate programming, the program can be created in a "stream of consciousness" order, which allows sections of the program to be developed top-down or bottom-up, whichever is most appropriate to the exposition of the program. The presentation of code which is irrelevant to the module being developed can be deferred until a more suitable point, focussing the author's attention more clearly on the design and exposition of the current module.

Knuth developed a tool, WEB, which integrates the documentation and code of a program into a single file. WEB provides macro-processing facilities which facilitate the "stream of consciousness" order of presentation. It was intended to make it easier for experienced programmers to develop and document programs. He quotes Anthony Hoare:

> "Documentation must be regarded as an integral part of the process of design and coding. A good programming language will encourage and assist the programmer to write clear, self-documenting code, and even perhaps to develop and display a pleasant style of writing."

The best known works using literate programming are Knuth's text formatter, TEX [67], and algorithmic font generator METAFONT [68]. These programs, along with their supporting utility programs, constitute a large body of software which was the reason for WEB's creation. The success of TEX has been due in some extent to the ease with which it has been ported to new machines, which in turn is due to it being written in WEB.

1.3 Syntax-directed editing

Syntax-directed or structured editing is also a relatively new field, which has waited for the technology to develop to make it practical. Syntax-directed editors restrict the possible operations to the structure of the document (usually a programming language) which is being edited. Motion and editing operations follow the syntactic units of the document. Templates are often provided, to insert new syntactic units or transform syntactic units into different forms. Documents created with syntax-directed editors have the property that the they are always syntactically *correct*, even though they may not be *complete*.

Syntax-directed editors incorporating knowledge of a programming language's semantics can detect and display semantic as well as syntactic errors, giving immediate feedback to the programmer developing the program. Semantic knowledge can be used to generate code incrementally while editing, avoiding lengthy recompilation delays.

The best known syntax-directed editors are the work of Reps and Teitelbaum. Their Cornell Program Synthesizer [109] was the one of the first practical examples of a syntax-directed editor. It was based around an incremental attribute grammar evaluator, which updated only the parts of the program representation which were changed by an editing operation. The Synthesizer Generator [97] is a successor to the Program Synthesizer, which allows the creation of syntax-directed editors from a specification written in the Synthesizer Specification Language (SSL).

1.4 Motivation

In the introduction to their book about the Synthesizer Generator [97], Reps and Teitelbaum say:

> "Programs exist not in isolation, but in relation to supporting documents. Whether interleaved with the program or separated from it, designs and specifications become large and unwieldy; these auxiliary languages themselves need language-based tools. By combining knowledge of both the programming language and the design or specification language in a single programming tool, it is possible to furnish programmers with support for creating programs according to particular methodologies."

There are several particular advantages that a syntax-directed editor for literate programs might bring compared to traditional programming tools (editors, preprocessors, and compilers). Chapter 3 will explain these points in more detail.

- The knowledge required to use a literate program editor will be less than the equivalent traditional system. In a normal WEB system, the programmer must know the programming language, the documentation formatting language, and the WEB system's language for controlling the interleaving of code and documentation. A literate program editor can present the documentation and code separation visually, removing the need to learn the control language, and the documentation language can be abstracted to generic markup operations for which templates can be provided without removing the possibility of accessing the full power of the formatting language if needed. Templates can similarly be provided for the syntax of the programming language, reminding the programmer of the possibilities available.

- Syntactic and semantic errors can be interactively checked and displayed in both the program code and the documentation. In particular, the problems associated with textual expansion in normal WEB systems are completely prevented.

- Semantic error checking provides a degree of debugging in the original literate program format, rather than the post-processed form output by some WEB tools. Incremental code generation and execution may be provided to allow completely integrated debugging.

- The benefits of publishing tools such as tables of contents, indices of identifiers, and module cross references are shown by some of the studies cited in chapter 2. Such tools may be provided interactively, with the added benefit of always being up to date.

The advantages which could be provided by a syntax-directed literate program editor can be summarised as reducing the effort required by the programmer. Mistakes are caught earlier in the development cycle, and high-quality information about the structure of the program improves comprehension, reducing bugs and shortening development time.

2

Literature Survey

2.1 Literate Programming

2.1.1 WEB

The term "Literate Programming" was introduced in Knuth's seminal paper [66]. This paper provides an introduction to Knuth's WEB system, and motivates the specific concerns of literate programming.

Knuth's original WEB system (as used for TEX and METAFONT) is a macro pre-processing system tailored for the Pascal programming language and the TEX typesetting system [65]. A single source file contains "modules"[1] which consist of a *documentation* part, a *definition* part, and a *code* part.

The code part of the module contains a fragment of program, which is associated with a name. References to this name can be made in other code parts, indicating that the fragment associated with the name is to be inserted at that point in the program. The features of WEB macros which make them useful for explaining programs are:

- More than one association can be made to each name — all of the frag-

[1]The choice of the term "modules" is an unfortunate one in retrospect, because WEB modules are not at all similar to the more common meaning of modules in Computer Science — that of separately compiled units of a program with a well-defined public interface and private implementation.

ments associated with the same name are collected together before expansion. This allows pieces of code which have syntactic constraints on their positioning in the program to be separated out and explained in juxtaposition to related pieces of code.

- The names of code parts in the WEB system can be whole sentences, containing nearly any sequence of characters desired. References to names can be abbreviated, as long as the abbreviation is unambiguous, for the convenience of the programmer.

Unnamed code parts are also permitted: these are collected together and used as the root module of the program, which will be expanded to create the whole program source code.

The definition part of a module is used for macro definitions which are expanded in the code parts. These macros are a single line of text, optionally with one parameter which is substituted in the replacement text. The definition parts of WEB were included mainly to make up for deficiencies in the Pascal programming language.

The documentation part of a module is intended as a commentary explaining what the code and definition parts are used for. There is a very rudimentary document structuring facility provided by "major modules", which are made more visually prominent.

There are two programs in the WEB system, both of which operate on the single source file: `TANGLE` takes the source file and produces a compilable Pascal program from it, and `WEAVE` takes the source file and produces a typesettable TeX document from it. The program listing produced by `TANGLE` is deliberately obfuscated by removing comments and non-significant whitespace, with the intention of forcing the programmer to use the typeset listing. The `WEAVE` program which produces typeset listings of the program incorporates a sophisticated, but unfortunately not very customisable, pretty-printer for the Pascal code sections. Knuth's ideas of good program layout do not suit everybody, and may be one of the reasons why programmers are put off using WEB. Automatic cross-referencing and indexing of program modules and identifiers in the program is also performed by WEAVE.

WEB does not make any attempt to hide the details of typesetting commands from the user, and indeed adds another (albeit simple) macro language to the systems that the user must master to produce elegant programs. A simple "changefile" facility is provided for merging system-dependent changes into the program. Changefiles define a list of paragraphs which should be taken out of the program, with altered versions which should replace them.[2]

While Knuth himself did not develop any further literate programming tools, he foresaw the possibilities for interactive tools such as source-level WEB debuggers and real-time WEB display tools.

2.1.2 The Literate Programming Paradigm

Some essential points of the literate programming paradigm which Knuth did not make explicit in his paper have been highlighted by Van Wyk and Thimbleby. The main point is called *verisimilitude* by Van Wyk [121] to indicate that *exactly* the same text is used for creating the human-readable documentation for the program and the compilable program. Thimbleby makes a good job of explaining the literate programming *paradigm* in his review of Lindsay's program [123]. He opines that in order for literate programming to qualify as *paradigmatic* it should provide certain features essentially for free; these feature would be expected to include flexible order of elaboration, cross-referencing, indices, typographical niceties, and mnemonic names.

Detig and Schrod [101] believe that there should be four characteristics for a programming system to qualify as literate:

Integration. Integration of documentation and program code in one document allows the user to see and change both parts at one time. There is no

[2]The idea is similar to the *context* `diff` found on UNIX systems today.

guarantee that this will happen, but there is more chance than if the documentation and code are separate.

Order of exposition. Very often a program text is arranged in a different way than it is developed by the author. Usually this is due to restrictions in the language syntax. A literate programming system allows the author to adapt the structure to the needs of other authors or readers instead of enforcing the needs of compilers.

Refinements. Named program parts with arbitrary names (not only identifiers, but whole sentences describing pre- and postconditions) supports both top-down and bottom-up program development.

Publication Tools. Literate programming systems should have tools to help the author in his presentation work. These may include index creation, table of contents, graphic subsystems, hypertext editors/structures, versioning, change marks, annotations, *etc.*

These criteria can be used to evaluate how well the literate programming systems examined in the rest of this section fulfil their function. Many of the simpler systems designed for programming language independence do not provide publication tools, which require a knowledge of the target programming language. A few systems do not provide flexible order of exposition; in a couple of cases this does not matter because of the nature of the programming language used.

2.1.3 Extensions to WEB

Extensions to the WEB system to make it more useful in “real-world” situations have been made by several people. Many of these extensions have been alterations to the change file mechanism, to cope with multiple changefiles.

Damerell [32] altered WEB’s program pre-processor (Tangle) to increase the level of error detection. Some difficult errors to trace in both the WEB macro language and the Pascal program code are detected. In addition, errors were reported to a log file as well as to the terminal.

Appelt and Horn [5] extended WEB to use multiple changefiles for programs with multiple system-dependent changes and revision levels. They point out that this facility is most useful when changes are orthogonal, but they do allow the possibility of prioritising changes which affect the same part of the original source file.

Both Guntermann and Rülling [48], and Sewell [103] wrote separate pre-processors to allow the use of multiple changefiles with WEB. They took different approaches to changefile conflicts: Sewell's approach in his WEBMERGE program was similar to Appelt and Horn's, in which one changefile took precedence over the others, and a warning of a conflict is issued. Guntermann and Rülling's program, TIE, applied each changefile successively, as if the previous changefiles had been merged with the original program. This approach is more suitable when changefiles are being used for development of new versions of a program, where the new code is developed as changefiles without altering the original program, and then is merged in to create a new release version. Both of these pre-processors create a new WEB file generated by merging the original program and changefiles.

Breitenlohner's PATCH processor [17] takes the same approach to merging multiple changefiles as TIE, but also has a file insertion feature; a control sequence is used to indicate that a patch is to be inserted in the generated WEB file. This feature allows common sections of code to be factored out of programs which use them, and maintained as one source file. PATCH has three modes of operation: in *merge* mode it creates a single WEB file and changefile, suitable for feeding to WEAVE and TANGLE. In *insert* mode PATCH produces one WEB file containing all changes and insertions, and in *update* mode all changes are applied to the primary patch, but insertions are not made; this mode is applicable for creating new release versions once all changes have been fully tested. Breitenlohner found the PATCH code useful enough to include in TANGLE and WEAVE, eliminating the need for an extra pre-processing step.

2.1.4 WEBs for other languages

There have been several adaptations of WEB to other programming languages, including C, Smalltalk, Modula-2, FORTRAN, Ada, C++, Scheme, and Reduce.

The first adaptation of WEB to another programming language was Harold Thimbleby's cweb system, created in 1984 [112]. This system was designed before Knuth's article in The Computer Journal [66] appeared — the author heard Knuth give a talk about literate programming, and implemented his own system from the ideas presented in the talk. Thimbleby used troff as the formatter for his system, but also included a cut-down formatter in the system for quick-and-dirty draft printouts.

Thimbleby's cweb system does not include the more sophisticated pretty-printing functions of Knuth's WEB, but does allow the programmer to lay the code out in the way he wants. Some manual intervention is necessary to create good-looking listings with Thimbleby's system. A simple alignment facility is provided to insert markers for aligning fields in the code listing, but no parsing of the program code is performed for pretty-printing. The lack of sophisticated parsing of the program code means that an index of identifiers could not be generated automatically.

Cweb detected when changes were made to the actual code in a file (not including the comments), and re-wrote the output file only when a substantive change was made, allowing `make` to work more effectively on these files. Included headers are supported by Thimbleby's cweb.

Thimbleby later reported experiences of using cweb [111], including subjective evidence that students forced to use cweb produced better programs and reports for assignments than those who did not use it. He states that:

> "In the past program listings were submitted for examination as subsidiary material, which was usually extensive, superficially homogeneous and disorganised in comparison with the written project report. The project report would also contain extensive reference to the program and often laborious internal documentation. Those students who have used cweb have presented informative programs

integral to the project report, and the previous tedious internal documentation is in one place and is easier to mark."

He examined the possibility of including literate programming systems in Integrated Project Support Environments (IPSEs), and mentions some of the possibilities for WEB manipulation tools, but did not pursue this line of work any further.

Three more adaptations of WEB for C were made by Guntermann and Schrod [49, 100], Levy [74], and Dong [128]. Levy's version translates WEB's definition sections into C pre-processor directives, which are then processed as usual by the compiler. He also introduced an "include file" directive, allowing CWEB programs to be split up into multiple files. The tangled output of Levy's version included `#line` directives which are recognised by most C compilers, allowing compilation errors to be referred directly to the line in the CWEB source file, rather than the intermediate tangled file. Levy's version of WEB for C has been developed by himself and Knuth to handle C++ constructs, and is becoming the de-facto standard WEB for C.

Dong's CDS literate programming system combines the C programming language with a language called SP, for typesetting Chinese and English.

Sewell [102] created a literate programming system for Modula-2, called MANGLE. The changes from the original Pascal-based WEB were easy enough that the implementation was done in changefiles. The MANGLE system is used for examples in his book about literate programming [104].

A specialised application of WEB for Modula-2 was demonstrated by Kredel [71]. He created a translator to convert routines for a computer algebra system into Modula-2 routines using WEB, with a view to using Sewell's MWEB if it is available.

A version of WEB for Ada was created by Wu and Baker [127]. Their conclusions after using it were that it was a far superior method of producing documentation, but there were some problems with WEB as implemented. In particular the extra directives required to produce good-looking code were awkward, the lack of hierarchical structure in the document frustrated good explanation of the program, the requirement for code segment names to be used before defined in WEB was not suitable for writing re-usable components and that explanatory code which is not part of the program cannot be formatted the same way as the code. These criticisms are echoed by some other authors, and may be due to the original WEB being over specialised.

Adaptations of WEB for Ada (again) [87], Scheme [89], Fortran [7, 8] and C++ [58] have also been done, with varying degrees of success.

Reenskaug and Skaar's adaptation of literate programming to Smalltalk is interesting, because it integrates the concepts of literate programming to work within the Smalltalk environment and idiom.

The Smalltalk environment allows integration of code and documentation as usual, but also allows easy inclusion of diagrams and figures in the documentation. An interactive "galley editor" gives a graphical overview of the structure of the document, with icons representing text, code, figures and pictures. Insertions, deletions, and re-arrangements can be performed on the document by manipulating the icons corresponding to sections, using the common cut and paste idiom found in many GUIs. The structure of the document can include nested sub-sections, to any level required.

Facilities are provided to browse the Smalltalk program library, and insert the code of objects into the editor. The code fragments in the editor can be compiled interactively, and placed back in the program library if desired. This does restrict the code fragments allowable in sections to compilable Smalltalk classes or methods. One of the "paradigmatic" features of literate programming (see section 2.1.5) is removed by this restriction; the ability to refine the program by using arbitrary names to represent code which has not been explained yet. In the Smalltalk context this is not a particular problem, as Smalltalk systems are usually built out of many small objects which communicate with each other, each object consisting of only a few lines of code. All of the classes and methods nested inside a particular section can be compiled at once, if desired. Sections later in the document can be used to over-ride earlier sections, allowing patches or variant versions of the program to be included with the original code.

The document formatting capabilities are provided by the Smalltalk system's normal methods, and do not need special attention or learning by the user.

The interpreted interactive nature of the Smalltalk literate programming system is more attractive than a traditional edit, tangle, and compile cycle, mainly because the overhead for tangling and compiling code is not necessary. Incremental tangling (and possibly weaving) of literate programs for traditional computer languages could be used to improve this situation.

Reenskaug and Skaar's experience of their literate programming system suggests that a good initial idea about what is wanted and how to do it is necessary. They suggest that literate programming is good for developing programs, because the assumptions and "soft" decisions built into the program become visible, but have reservations about its use during the initial experimentation or final testing of programs, where the specification or details of the program may be changing rapidly.

2.1.5 Language independent WEBs

The close tying of Knuth's WEB to the Pascal language may have been one reason for the lack of interest in literate programming; it has been seen, in the context of Pascal, as good for teaching but not useful for real-world programming. There have been several systems created which allow any programming language to be used, but this flexibility comes at a cost: these systems do not pretty-print the code sections, and more importantly, they do not include the index of identifiers which is so useful in WEB programs. These features have been described as *paradigmatic* by Thimbleby (in his review of Lindsay's file difference program [75]), to indicate that they are part of what makes a literate programming system *literate*. Thimbleby suggests a set of features for literate programming systems which should be essentially effortless for the programmer to use.

Hanson [52] used a simple literate programming system called `loom`, which was originally written by Incerpi and Sedgewick for use in Sedgewick's book *Algorithms*. Text and program are intermixed, with text before and after program fragments. Loom is implemented as a preprocessor which extracts the fragments of program and text, pushes them through a system of filters, and integrates the output. Most of the indexing, cross-referencing, and pretty-printing facilities of WEB are not provided by loom, but it does have filters to create an index of identifiers. Hanson's example includes replacement code introduced after profiling which generates conditional code in the output program.

Williams' FunnelWeb [125] (named after a particularly nasty antipodean spider) is a literate programming tool in the style of Knuth. Instead of two programs to extract the source code and typeset listing, FunnelWeb provides one program to perform both functions. FunnelWeb does not contain knowledge about the target programming language (in fact output files do not need to be any computer language — they are treated as free text), so it does not perform the pretty-printing and indexing of WEB. It does contain knowledge about the target formatter, but it normally neutralises formatter commands, typesetting the documentation exactly as it appears to the user. Special control sequences are provided to provide simple formatting capabilities, which can then be translated for an appropriate formatter. There is an escape mechanism to allow use of formatter control sequences if necessary. FunnelWeb forces the user to declare if a piece of code is being added to another module; this is at odds with most other literate programming tools, where the typeset listing will indicate if a module is being extended, but the input syntax is the same for all similarly named modules.

FunnelWeb includes a scripting language, which can be used for processing multiple files, writing utilities for FunnelWeb, and regression testing. The FunnelWeb shell was written so that FunnelWeb utilities did not have to be re-written in the plethora of command languages available on different systems. A suite of scripts are included with FunnelWeb to perform automated regression testing.

FunnelWeb has an option to provide a similar facility to Thimbleby's cweb with respect to Makefiles. If the files written out are identical to previous versions, the previous versions are left unaltered, allowing minimal re-compilations to be performed easily. Williams is considering modifying FunnelWeb to read files containing pretty-printing instructions for the code parts of documents.

An alternative approach to creating literate programming tools is exemplified by `noweb` [92]. Ramsey built a minimal literate programming system, in the style of early UNIX programming tools, which supports a syntax very similar to WEB. The `notangle` and `noweave` tools convert the input syntax into an intermediate representation which is easy to manipulate using common UNIX tools. Shell scripts are used to compose the tools which convert the input syntax to intermediate representation and intermediate representation to final output. The system as implemented is simple, and does not provide pretty-printing or automatic identifier indices, but the modular construction makes it easy to insert specialised tools for particular languages where required. The `noweb` system is nearly formatter independent, having about 30 lines of AWK which convert the intermediate format for TEX or LATEX. `noweb` supports multiple programs in each source file, allowing the user to specify which root node will be expanded by `notangle`. `noweb` also provides a tool to convert the source files into a normal program (like tangling), but with the documentation included as comments, so that programs written under `noweb` can be removed from `noweb`'s control easily.

Ramsey's work is also interesting because it includes the only published case of multiple programmers using a literate programming tool to work on a single program [93]. A language-based editor (Penelope) intended to help programmers formally verify Ada programs was created using a system based on Levy's CWEB. After three years of work, the source document for the editor was over 33,000 lines long, of which about 13,000 are documentation. Seven programmers have worked on the project during that period, with up to four working concurrently. Ramsey likens a literate program that is being extended and maintained to a car repair manual, rather than the polished novel which Knuth's finished works resemble.

The literate programming system proved inadequate in some areas; describing data structures was hampered by lack of support for diagrams or pictures; tables and figures were difficult to present; the pretty-printer's choice of line breaks and indentation were disliked. The output of `WEAVE` required tedious changing to include in other TeX documents, and was even more awkward to include in LaTeX documents. The marked difference between the appearance of the WEB source and the typeset listing was a source of confusion. Auxiliary tools to extract parts of the WEB program were written, so that small parts of the source could be printed without the paraphernalia of large indices and tables of contents.

Ramsey concludes:

> "We cannot say to what extent literate programming can replace standard software development methodology. However, putting a clear description of design in our source code helped a changing team of programmers to develop it over a span of three years . . .
>
> We believe that literate programming helped us substantially. This belief is based not on measurements but on our subjective comparisons of experience on this project to other projects. A programmer who has used standard software development systems at an international computer manufacturing company reports that a key difference in Penelope was that the documentation was *used*, precisely because of its proximity to the source code."

Briggs's NuWeb [18] is based on the ideas of FunnelWeb and noweb. It allows the use of many programming languages and creates multiple output files. LaTeX is used as the output formatting language rather than TeX, so that the advantages of hierarchical section structure and support for pictures, bibliographies and cross-references can be used. Code is not pretty-printed, and the layout is left up to the programmer. Indices of file names and macro names are provided, but not identifiers. NuWeb is a single program that creates the formatter and language files in one pass.

The VAMP system for program refinement [117, 119] has been in use and development since 1982. The documentation and code sections of literate programs are called *stubs* and *texts* in VAMP. Stubs are a set of contiguous lines starting with a start-stub command, and ending with an end-stub command. Slot commands may be included in stubs to indicate positions where other stubs are substituted during code expansion. A feature which is not found in any other literate programming system is that slot commands may contain an indication of a part of the current stub which is to be omitted if another stub is substituted for the slot. If this facility is used properly, refinements can be compilable and executable at all levels of expansion, and temporary declarations may be introduced to make stubs compilable, but replaced at a later stage of development. The VAMP system was developed on VAX/VMS, using the RUNOFF document formatter to typeset source listings.

A criticism of VAMP is that it does not insert typographic markers to indicate where slot commands are, or even which stub is which. In a complicated document with many refinements and omitted slot texts it is thus difficult to understand the actual structure of the program. VAMP is intended to be more or less formatter independent, so it is not obvious how slots and stubs could be marked. VAMP does not provide the pretty-printing or identifier indexing of WEB, another consequence of the goal of formatter-independence. VAMP does allow creation of more than one output file from one or more input files; the output files are not limited to programs, and can include data, tables, *etc.*

CLiP [116, 118] is a successor to van Ammers' VAMP system for literate programming. CLiP takes a unique approach to formatter and language independence which allows it to be used with almost any language and formatter or word processor. The refinement modules and references (termed stubs and slots by van Ammers) are indicated by specially distinguished programming language comments. The particular form these comments take can be tailored to the languages used by changing run-time parameters of CLiP. This approach requires the programmer to use a programming *style* rather than learn an extra set of directives to drive the literate programming tool. Also, any formatter or word processor which the programmer is familiar with can be used to create the program, reducing the learning overhead for CLiP.

Documentation sections of CLiP programs can be formatted using the facilities allowed by the formatter, including pictures, graphs, tables, or whatever ancillary material is required. The code sections are formatted by including them in the word processor or formatter's "literal" or "verbatim" mode. The code sections of CLiP programs are not fully parsed by CLiP, so no automatic indexing of identifiers can be performed. Cross-referencing between sections can be done manually using whatever facilities the formatter or word processor provides.

CLiP can extract multiple output files from one or more input files. Top level stubs are marked with a directive giving the name of the output file. One stub is normally required to replace each slot, with the replacement done on a line-by-line basis. There are several options which can be used to change the behaviour of the stub/slot replacement mechanism. These are:

multiple This option indicates that multiple stubs may be provided for the slot in which the option appears. The stubs are concatenated together in the order in which they are found. This is similar to the behaviour of WEB, but WEB does not allow the possibility of rejecting multiple definitions for a refinement.

leader Multiple slots may have one *leader* stub defined, which will be inserted at the front of the multiple slot only if there are any other stubs provided for the slot. This facility can be used to insert required keywords in front of syntactic placeholder slots (such as constant or variable definitions).

quick Short stubs can be defined using the *quick* option, which terminates the stub at the first blank line, rather than requiring an end of stub comment.

comment off The comments which delineate stubs and slots are normally extracted with the code sections. This behaviour is undesirable in some cases, such as when data files are incorporated into the literate program. The *comment off* option prevents the stub and slot comments from being included in the output file. This option can be used locally to suppress the information about the slot in which a stub is inserted.

optional CLiP provides a mechanism in which slots can be omitted completely, for example for debugging information which is not used in production versions of the code. If a slot contains the *optional* declaration, an error will not be flagged if a corresponding stub does not appear. A separate input file can contain debugging stubs which will be included if the file is specified on CLiP's command line.

default The *default* option is useful for porting and tuning code. If a stub contains the *default* option, it will only be used if there is not a normal stub which matches the same slot. Portable versions of code can be declared in default stubs, and overridden by optimised versions in machine-specific input files, or error messages for unimplemented functions can be put in default stubs, and overridden by proper stubs as they are written. This facility allows CLiP to be used as a design tool, to create the overall plan of the program and flesh out details later.

More than one of these options can be specified in one stub or slot.

CLiP and VAMP were designed as stepwise refinement tools, rather than specifically literate programming tools, and the facilities of these tools (such as stub redefinition) are weighted to this end. (CLiP stands for *Code from Literate Programs*, emphasising its code extraction rôle.) Whilst CLiP does not provide the full facilities which might be expected of a literate programming tool, it is the only completely formatter and language independent literate programming tool to date.

2.1.6 Multilingual WEBs

Another approach to adapting WEB to different languages was taken by Ramsey's Spidery WEB system [90, 91]. In this system, the user can build specialised WEBs for different languages by creating a description of the language and processing it with the Spider tool. The class of language which Spidery WEB can handle is restricted to languages which have LR(1) grammars (Spider uses `yacc` to parse the scraps), and have similar lexical properties to C or Pascal. The description file contains a partial grammar of the language, and also defines translations of the language tokens for pretty printing (*e.g.*, the token `!=` could be defined to pretty-print as $\neq$), and the token categories (which are used to alter the operator spacing to emphasise precedence). Spider provides facilities to insert compiler directives to keep track of the source code line numbers, if the language supports them.

Ramsey's Spidery WEB system has been used to build WEB systems for C, Ada, SSL, AWK, LARCH, and Reduce [47]. A version of WEB for Maple may soon be available from the authors of the Reduce WEB. The Spidery WEB system makes the need for continual adaptations of WEB to yet-another-language redundant for the most part, but does not add any new ideas to the literate programming field.

Krommes's FWEB [72, 73] is the archetypal "all but the kitchen-sink" development of Knuth's WEB. Not only does it provide a literate programming tool for Fortran-77, it also manages to support Fortran-90, C, C++, TEX, and has a powerful enough macro processor to translate Ratfor and some M4 commands, *within the same document*. This is the only language-specific system which supports more than one language concurrently. Just about every aspect of the documentation and code layout for FWEB can be customised, including the colour of the text! FWEB input files are still structured very much like Knuth's WEB — the code is derived from Levy's version of CWEB, which in turn was derived from Knuth's WEB. The major modules of WEB which indicated important divisions between parts of the document have been expanded to include a hierarchy of possible section types.

The language pretty-printing and identifier indexing information in FWEB is built into the FWEB code, rather than being read from separate tables. This would be an obvious extension to many of the WEB systems presented so far; it is surprising that it has not been implemented yet. FWEB supports LATEX as a document formatter as well as Plain TEX; some of the facilities missing from WEB (such as documentation cross-referencing) can be thus be provided if the user is proficient in LATEX.

FWEB must count as the most capable example of a literate programming tool based on WEB. However, the future path for development of FWEB is unclear — how long new languages and customisations can continue to be added before the complexity of the program becomes overwhelming is uncertain. It is probably a testament to the power of literate programming to present problems effectively that FWEB has managed to reach its current state, with only one person driving the development.

2.1.7 WEBless WEBs

Fox [45] created a "WEBless literate programming" tool for C, called c-web (pronounced "see no web"). The implementation of c-web is clever; a comment at the start of the program contains an `\input` statement which TEX uses to input a macro package which implements the special formatted treatment. All c-web commands are contained within normal C comments. The C compiler ignores the comments, but TEX reads the macro package and typesets the program according to the c-web commands. The advantage of the method used by Fox is that the program is directly compilable using a normal C compiler, and can be formatted using TEX, with no other pre-processors. However, c-web is in effect just a C pretty-printer, lacking any of the refinement features which make literate programming interesting.

The `doc` style file [80] for LATEX performs a similar task to c-web for LATEX style files, allowing documentation and code to be intermingled in the same source file. Again, the refinement steps are not provided, but output to multiple files from a single source is provided. The latter facility is used to write driver files and indices which can be run through LATEX or MakeIndex as appropriate.

A variation of WEB-less literate programming is the inverse comments used by some implementations of the lazy functional programming language Haskell [57]. In Haskell, lines which do *not* begin with '>' are treated as comments. This allows Haskell programs to be inserted in other documents and still be executable. The re-ordering performed by WEB is not necessary in a language like Haskell, where names can be used before they are declared and functions tend to have small bodies which can be explained in one unit, as the program can be written in just about any order desired anyway. Publication tools such as pretty-printers and cross-references are not provided by the Haskell compiler, but could be written as separate tools.

2.1.8 WEB tools

Several tools to facilitate literate programming have been created, but nothing as grandiose as the integrated environment or IPSEs that Knuth and Thimbleby envisaged.

An interactive interface to WEB for the GNU Emacs customisable text editor was written by Mark Motl at Texas A & M University [81]. This gives the emacs editor a limited facility for editing and compiling WEB programs. The operations provided include movement by modules and major sections, creation and deletion of modules, and selection of modules from a simple table of contents and index. The WEB mode is driven by menus, which can be invoked by binding Emacs commands to keys.

A more sophisticated interactive literate programming system was proposed by Brown [23–26], and partially implemented. The design of the literate programming environment was based around a WEB editor, with a source-level WEB debugger, a WEB importer, high-level language importer, personal preference database, and a control panel to guide the action of the other parts of the system. The principles guiding the design of the literate programming environment were that it should bring the advantages of the typeset program listing to the screen, and that it should automate many of the common tasks involved in WEB programming or provide utilities which ease significant portions of the literate programmer's work.

In practice, the former principle means that the automatically-generated information which is available from the WEB system (such as a complete index, list of named module, and module cross-reference list) should be available within the editor. The program should be displayed without WEB commands in the source interfering with the comprehension of the code. It was not thought necessary to have exactly the same display on the screen as the typeset listing provided, but it was noted that this would be best to increase the reading and comprehension speed to the same levels as the listing.

A prototype of the WEB editor at the heart of the system was built, and studies of student programmers were performed using this editor. The WEB editor included a hypertext-like interface to WEB, where the relationships between various modules were used to provide instant links from one module to another. The relationships between modules which were deemed useful were:

1. Module *n* uses module *m*. If a module includes another module, or is included by another module, a button appears on the screen allowing instant access to the other module.

2. Module *n* and *m* both appear in the same index entry. Related modules may not necessarily be direct ancestors of each other. Buttons were displayed to access other modules which were categorised under the same topic in the index.

3. Module *n* is an addition to module *m*. Modules which augment other modules were made instantly accessible from each other.

4. Modules *n* and *m* both use the same variable. This case was used to allow cross-referencing between the definitions and uses of variables. There does not appear to be any reason to restrict this sort of cross-referencing to variables rather than procedures, functions, and other symbols, but Brown mentions only variables. This may just be an oversight.

The WEB editor also includes an index display and a graphical representation of the program, which the user can browse to get an overview of the relationships between modules.

The students' performance on typical maintenance tasks was evaluated when using either the WEB editor and a normal listing of the program, or a normal text editor and a (normal) listing of the program. An experiment was performed to measure the difference in code comprehension between the methods mentioned above, using a display-only version of the WEB editor. Tasks were given to the students which involved identifying the places at which changes needed to be made to accomplish a goal, and to suggest (in English) what changes needed to be done. The results were marked on the correctness of the places which were identified, and the accuracy of the changes suggested, as evaluated by an experienced literate programmer who had performed the same changes on the programs involved. The difference in comprehension between the methods turned out not to be statistically significant, but there was a correlation between the resources used and their performance. The students who performed best when given the standard editor and listing tended to use the listing almost exclusively. The same students, when given the WEB editor and a listing, tended to use the WEB editor in preference to the listing.

A questionnaire was given to the students in the period after the experiment had finished, in which they were asked to rate editors and the listing for ease of use and comprehension of the code. The ratings were similar for the listing and the WEB editor, but significantly worse for the standard editor. A further question asked whether the students felt they would do the exercise better with the WEB editor than the standard editor, if they had to do it again. The WEB editor was unanimously chosen.

It is not clear from Brown's thesis whether all of the features in the design plan for the WEB editor were implemented in the prototype editor. Some of the facilities which were probably omitted were:

1. Change file editing. The design for the editor included the facility to alter the literate program, and have the modifications saved as a change file, rather than as alterations in the original source. This would ease the use of changefiles significantly, and allow many programmers to work on the original source simultaneously, without interfering with each other. The changes could then be integrated into the master copy using a utility like Guntermann and Rülling's TIE, or Sewell's WEBMERGE.

2. Graphical representation of data structures.

3. Data flow diagrams. The plan for the editor included the ability to create diagrams showing how and where variables were used. This facility was intended to include aliases for variables created using WEB's macro definitions.

4. Syntax directed features. The editor was intended to incorporate some syntax-directed features for the code and documentation sections, such as template insertion. The prototype editor was syntax-directed with respect the WEB syntax, automatically handling the division into modules, control texts and indexing. This part was designed so that the user never needed to enter a WEB command directly, but experienced users could type WEB commands if desired.

The WEB editor was the only part of the literate programming environment which was implemented. The design of some of the other parts of the system was decidedly incomplete, and read like a "shopping list" of nice features, with little attention to feasibility.

The important part of Brown's work was the evaluation studies that he performed using his prototype WEB editor. These indicated that the WEB editor was a useful tool for aiding the comprehension of literate programs, and was in fact preferred to the program listing, even though the comprehension ratings were similar. Brown concludes that the full literate programming environment is worthwhile developing. The WEB editor is the part of the design which is most radically different from normal programming tools, and the other parts of the system should be minor improvements on the normal programming tools, if not particularly novel.

An interactive graphical literate programming tool called HSD (for Hierarchical Structured Document) was created by Tung [115]. HSD provides facilities similar to WEB, with an interactive graphical interface.

HSD sections can be hierarchically structured as a directed acyclic graph, rather than WEB's flat section structure (sections can also be included in more than one other section). HSD sections can contain comments, code, names of included subsections, and *define-box* and *add-to-box* commands, in any combination and order. The latter two commands are used to defer elaboration of code until later in the document.

HSD uses a Graphical Document Descriptive Language (GDDL) internally to represent the literate program, and generates the code or document by traversing the GDDL in different ways. The document is generated by a simple pre-order traversal of the GDDL, decorating the output with formatting commands where necessary. The code is generated by a two-pass algorithm which collects the deferred code together, and then traverses the graph again, outputting code with deferred code and refinements inserted at the correct places.

HSD was developed for Macintosh computers, using a commercially available set of interface tools. Tung does not provide details of the interface to HSD, except that it uses standard Macintosh text editing facilities.

A perspective on using WEB and other literate programming tools was presented by Bart Childs at the 1992 TEX User's Group conference [28]. The literate programming tools created at Texas A & M (mostly under Childs' supervision) were reviewed; these include:

- Mark Motl's WEB-mode for GNU Emacs,
- a WEB program module size histogram creator by Mamoun Babiker,
- a WEB statistics gatherer by Mark Gaiter,
- a change file analyser,
- a WEB structure viewer by Kevin Borden, and
- a Makefile creator for literate programs by William Needels

The data from the histogram creator is interesting, suggesting that a typical WEB program has more than 90% of the modules in a program under 25 lines long, *i.e.*, one screenful of a typical terminal. This data suggests that WEB programmers break the program into chunks which can be seen all at once. Childs also presents some data to suggest that van Wyk's characterisation of literate programmers as creators of their own programming systems is changing, and that there are many more users than creators now.

The Cnest and Cscope tools described by Cordes and Brown [30, 70] can be used to examine WEB programs. Cnest illustrates the location and nesting level of a current module within the overall scope of the program code. Cscope displays the program and the modules that constitute it in an easily understood form.

Brown and Czedjo proposed a hypertext representation for literate programs which transformed the hypertext linkages into relational database queries [21]. A simple prototype of this system was constructed, to show how the mapping from hypertext linkage to database query was performed.

Bishop and Gregson's `LIPED` system [16] is a literate program editor which is designed to let the programmer work in a format which closely resembles the final output. It is aimed at minimal hardware systems (*i.e.* IBM PCs). Five views of the program are provided; *literate program*, *code*, *contents*, *cross-reference*, and *header*. Documentation structure is confined to major and minor headings, which are inserted in the table of contents automatically. Linkage functions between views are provided, allowing the programmer to jump between corresponding parts of different views quickly. `LIPED` uses the first fragment name encountered as the root fragment. Reserved words are highlighted in the compiled code, and the highlighting is suppressed between string quotes, but not in comments. The table of reserved words can be loaded dynamically, giving a rudimentary pretty-printing facility. A data structure mapper will be included as a separate view in later versions.

Smith and Samadzadeh [105, 107] describe a tool called `WEBmeter` for automating software complexity measurement. This tool takes data for Halstead's Software Science measures, McCabe's cyclomatic complexity number, and some measures specific to the WEB environment. Some hand calculations were also performed on small WEB programs for Yau and Collofello's design stability measurements. A large amount of data was gathered from several literate programs, but no real conclusions were formed; the work was seen as a pre-experimental study for forming hypotheses about WEB complexity and stability. They make some personal observations about the use of WEB, indicating an increase in understandability noticed for large programs, and an increase in tedium for small programs.

VistaTech's HyperWeb [43] was developed to combine hypertext technology with Knuth's WEB methodology. It is one of the few literate programming tools to be exploited commercially, in the form of the PCTE Workbench product. HyperWeb is an hypermedia-based software development environment which supports development and maintenance activities. Software is modelled as a web of small components that reflects its natural design rather than the constraints of the programming language. Complex relationships between the various software artifacts comprising a system (*e.g.*, requirements, designs, specifications, code, test scripts, configurations, *etc.*) are captured and represented explicitly. Frequently this knowledge exists only in the minds of the individual developers working on the software, and maintenance programmers spend much of their time trying to recapture this knowledge. HyperWeb tries to capture this knowledge so that it will not be lost when people leave a project. The system supports not just text, but documents of any sort, including diagrams, pictures, and even voice annotations can be linked into the web.

Annotation, decomposition, and refinement operations provide support for restructuring and documentation of the software artifacts. Annotations can be added to software artifacts to capture designer knowledge or to add comments for on-line code inspections. Decompositions are small conceptual units within a larger software system. Each decomposition focuses on clearly presenting its logic and omits irrelevant details. The relationships between these decompositions represent the natural structure of the system design. The decompose operation allows existing software to be broken up into a smaller units during maintenance, whereas the refinement operation allows decompositions to be created during new development.

HyperWeb augments the programming facilities provided by Unix, using the standard tools rather than replacing them with "better" tools that the programmer will not make the effort to learn properly. The tool integration framework provided by HyperWeb enables an existing set of analysis, design, and development tools to be integrated, making it easier for programmers to change their work habits to use the system.

The Igor project at Carnegie-Mellon University [40] is a new project to build a development environment for "hypercode". Hypercode will represent the program as a complex data structure linking together routines, class definitions, comments, specifications, diagrams, test code, edit histories, configuration info, and more. The programmer will be able to view and browse this hypercode at many levels of detail, and the code definitions can be presented in whatever order makes the most sense at the time. An extensive library of classes and functions will also be available, with librarian software to guide users in finding what they need.

2.1.9 Related work

The journal *Communications of the ACM* featured literate programming in Jon Bentley's *Programming Pearls* column twice in 1986 [14, 15], and then ran an infrequent column about literate programming from 1987–1990, moderated by Christopher van Wyk [33, 122]. These columns covered a variety of literate programming systems and techniques, using example programs written to solve problems set by the column editor. The programs were reviewed by guest reviewers, and these comments on their suitability for their purpose and clarity of exposition are perhaps more useful than the example programs themselves. It is only because the programs had aspirations to *literacy* that they could be reviewed in this way, and that the reviews did not turn into either marking an exercise, or a conflict of opinion about the problem's best solution.

The column stopped in 1990 because the moderator was concerned that the only people who seemed to be writing articles were implementors of literate programming systems, and that a fair conclusion to draw would be that one must write one's own system before being able to write literate programs. This conclusion is an illustration of one of the problems with literate programming; it may help create *better* programs, and thus reduce the amount of maintenance that a program requires, but the initial effort in writing a literate program is greater than the effort in writing a program using traditional methods. Users are reluctant to change to a system which requires more effort, even if the long-term gains are worthwhile.

The work of Oman and Cook [83, 85, 86] in program layout and formatting can be interpreted to provide empirical justification for research into literate programming. They performed experiments which showed that typographical re-formatting of code can significantly increase program comprehension. They call the set of principles which they used to re-format code the "book paradigm", because they presented their code listings in the format of a book, with all of the expected features of a book — pagination, table of contents, index, chapters, and sections. Typographical features were used to distinguish between externally and locally defined identifiers, procedure calls, and keywords. A key similarity with the literate programming paradigm is that the same source file was used to create the executable program, and to generate the formatted hardcopy listing.

Their research was founded in studies of programmer's behaviour when performing common maintenance tasks; Oman and Cook [86] state that:

> "All programmer comprehension studies support the existence of:
>
> 1. Mental schemata or plans that guide the programmer's comprehension of code. Programmers acquire and modify these plans through experience; they are an integral part of long-term memory.
> 2. Chunks or meaningful units of information that programmers use to organise and remember code.
> 3. Beacons or highlighted semantic clues that are used to direct the review and recognition of code. Beacons are used for searching, chunking, and hypothesis checking.
> 4. Multiple strategies and access paths used by programmers when working with non-trivial programs. Strategies are guided by a variety of plans and conjectures depending upon individual differences, application domains, and the implementation of the code and supporting system.
>
> "

Their work in typographical re-formatting of code tries to take these factors into account, to highlight beacons, and assist with multiple access paths by providing tables of contents and indices; chunks may be distinguished by insertion of blank lines.

The typographical program re-formatting of Oman and Cook incorporates some of the elements found in literate programming—generation of table of contents, cross reference tables, and index; division of the program into sections; use of proportional width fonts, boldface and italics in formatting code to distinguish different syntactic classes. In some areas they go further, by basing the program re-formatting performed on program comprehension studies, thus producing a measurable increase in program comprehensibility.

They conclude that "Good typographical formatting reflects the underlying structure of the code by providing visual clues and a variety of ways to view the code, which in turn aids maintenance activities. . . . Think about writing a book, not just a program."

Baecker and Marcus [9] used graphic design principles to make program source text more readable, understandable, and maintainable. The basic design principles from which they worked were:

1. *Typographic Vocabulary*: distinguish tokens by use of small number of appropriate typestyles.

2. *Typesetting Parameters*: adjust text size, word spacing, headline size and usage, *etc.*, to enhance readability.

3. *Page Composition*: use grids, rules, and white space to bring out program structure.

4. *Symbols and Diagrammatic Elements*: integrate appropriate symbols and diagrams to clarify essential program structure.

5. *Metatext*: augment source code with commentaries and mechanically generated supplementary text.

They developed a *visual compiler* called SEE, which processes the source code into an appropriate form for printing. Their method is similar to literate programming in that exactly the same text is used to create the printed copy of the code and the compiled executable program. The SEE compiler was built for the programming language C, and was based around the front end of the Portable C Compiler, modified to retain comments and preprocessor declarations in the syntax tree.

Their use of the *typographical vocabulary* for program presentation is based on the function of the token. Reserved words are not emphasised by using bold face, as most pretty printers (including WEB's) do; such strong visual effects are better used for distinguishing important classes of tokens (such as global variables). Varying word spacing and kerning can also be used to visually emphasise operator precedence. In certain cases, such as the presentation of structure declarations, the SEE compiler substitutes diagrammatic markers for the original syntax, making the program structure visually accessible.

No empirical evaluation of Baecker and Marcus's methodology is presented, but some subjective evidence for its usefulness in the form of comparative examples with processed and original source code is given. Several ways in which each of the design principles above can be applied to program presentation are discussed, and alternate representations of program fragments based on the same principles are shown.

Arab [6] implemented a program re-formatter for the Pascal language which also automatically prompted for documentation, using keywords appropriate to the syntactic units which were being processed. The documentation was inserted into the program as comments at appropriate places. The re-formatting style was based on the program comprehensibility studies examined by Arab, and was customisable within certain limits. The program was intended to make presenting and documenting programs easier.

Anand [4] started from the same concepts as Knuth, but worked to a different conclusion. He argues that the name of every unit or abstraction in a program should be a functional description of the abstraction. The naming method which he suggests is applicable to the modules of a literate program. Four criteria are given for functional descriptions:

1. The details of the entity must be predictable from the description.

2. The description must convey the function of the entity, (what it does) not its logic (how it does it) or context (where it does it).

3. The description must be as concise as possible within the previous constraints. A length of 6 to 20 characters is seen as acceptable.

4. Each entity must perform one and only one specific function.

Anand shows how naming by function enhances partitioning of modules and understandability. In this system, imperative phrases should start with verbs, pointer names are adjectival phrases. Boolean variables should be named so that the phrase "How to find if *variable*" makes sense; similarly pointer variables should be named by making sense of "How to locate *pointer*" and constants and other variables should be named by making sense of "How to find *constant*". Functions and procedure declarations use "How to *name*".

Anand proposes that the identifier naming rules for programming languages are enhanced to allow non-significant whitespace to improve the functional descriptions of elements.

Thimbleby presented and discussed some ideas for interactive editing of literate programs [111], before any interactive literate programming systems had been created. His comments are still relevant, in that the lessons which could be drawn are still being re-learned today. He suggested that WEB control sequences could be abstracted out by graphic conventions, such as displaying modules with frames to delineate them. Other possibilities mentioned are to allow the programmer changes in perspective, so that they can see an unobstructed view of the program or documentation, with the other part hidden away. He pointed out that free-text macro processing, as provided by WEB, clashes with structure editing, because the bodies of macros do not need to be syntactically well-formed.

Thimbleby suggested that the work of Feiner, Nagy and van Dam [42] on orientation cues is relevant to such an interactive literate programming editor.

The editor should support the following:

Annotation. A method so that the user can add notes "in the margins". These notes should be preferably handwritten, drawn or otherwise distinctive.

Folio. A standard display format which includes explicit information capturing the progress of the user dialogue, such as the current time and an iconic representation of the last folio.

Timeline. A canonical representation of the user's actions, probably shown in miniatures of folios, over a period of time, ordered left-to-right to show their sequencing.

Index. Various facilities so that the user can locate folios by abstractions.

Neighbours. The ability to view adjacent folios to the current folio, either by miniaturisation of all folios or using a "bifocal" method.

Colour. As an extra cue, perhaps to indicate the permanence of changes in perspective or to facilitate cross-referencing index entries.

Another valid point to consider is the amount of typographical control required of the program's author. Thimbleby contended that:

> "There is a valid school of thought (exemplified in the philosophy of SCRIBE) that author/programmers need not bother or waste their time with typography. Professional literate programming systems must provide an 'author' mode without access to typographical niceties, and without the learning effort (and temptation)."

In this vein, Thimbleby also considered the potential for language and formatter independent WEBs. He concluded from his own work that troff was an inappropriate choice of formatter, because of the poor programming interface. He also made the point that syntax analysis of code simplifies the formatting to be done by the user, because most of the code can be pretty-printed automatically.

Mitchell [79] tried developing a literate programming methodology based on data abstraction and a formal specification language. He also tried to provide a framework for what the programmer should write and where to write it. He used the programming language Modula-2 with the specification language OBJ, and some semi-formal "glue" of his own to tie them together.

The strength of Mitchell's work is not in his approach to literate programming, but in his view of the whole program development process, which leads to the presentation of the program in literate form.

He clarifies the concepts of *design* and *specification*, introducing the notions of *internal* and *external* design. In Mitchell's world, design is a process which involves choices; specification is the result of a design process. The terms internal design and external design are used to separate the choice of mechanics for the program implementation with the choice of higher-level interfaces and functions of the program.

Mitchell also introduces a model of development space which is interesting to review in the context of literate programming. His development space has three orthogonal directions; level of detail or abstraction, level of precision, and aspect coverage.

> "During the development of a program, a programmer can say more about the program in three ways; by becoming more precise, for example, by taking something expressed in an informal language and expressing it in a formal language; by considering more aspects of the program, for example, by starting to discuss a module not previously discussed; and by adding detail, for example, by showing how the operations on an abstract data type become procedures, thereby introducing the detail of side-effects."

Mitchell formulates a set of requirements for tools to support literate programming, based on the experience of developing the "literate" program presented in his thesis. He believes that the tools for the support of literate programming should be based on the principle that things are only defined in one place, and the same definitions are carried automatically to any other place at which they are needed. This requirement can be satisfied by the macro expansion facilities of WEB.

Mitchell states that the class of tools to support the presentation of programs can be divided into two sub-classes, which present text and diagrams. He also believes that there should be tools to present the results of analysing the program, as well as tools that prepare documents containing commentary and code. The principle of providing analyses of programs should form the basis of tools that present diagrams.

Mitchell's last requirement is that tools allow the programmer to make changes to polish programs as their understanding of the problem and program grow.

He cites Peter Naur's ideas on programming as theory building, as a model of what the documentation of a program provides:

> "We can regard what the person who writes a program knows that is not in the documentation of a program as a theory of the program. Loosely speaking, a theory of a program is all the true statements about the program; the documentation of a program is a theory presentation in that it contains enough of the true statements about the program that all other true statements can be inferred from it. What the original programmer can do more easily than another programmer is apply appropriate rules of inference to derive theorems about the original program and programs related to the original program."

The problem with Mitchell's approach is that he defeats the purpose of literate programming in two major ways.

1. The framework which he uses to specify what should be written where is too constrictive, and does not allow the order of exposition to be changed to suit the audience for the program.

2. No support tools are provided to assist with the integration of documentation, specification, and program, or to check the program against the specification. The programmer must do everything by hand, which results in a greatly increased work-load.

Brown and Cordes have argued that the design as well as the implementation of a program should be documented in the source code [22]. They showed examples of how the design can be integrated using stepwise refinement, data-flow diagrams, and the Jackson System of Development. They believe that WEB permits the actual design to be generated in a programming language, providing a direct link between design and implementation

Brown and Cordes also compared the literate programming methodology to the use of program design languages for the design stage of the software lifecycle [20]. The requirements for program design languages were examined and contrasted with what literate programming tools can provide, with the conclusion that literate programming tools provided all of the facilities required, and had the advantage that the modules can be expanded into a fully working program.

Parnas [88] also believed that documentation should be the primary function of programming. He described an ideal design process in which the documentation, and the consistency and completeness of the documentation, plays a major role in guiding the design. Who should write which bits of documentation was discussed, as well as which bits of documentation should be written retrospectively, if they were not written during the actual design process.

One of the objectives of the UQ2 program editor designed by Welsh *et. al.* was assistance with creating *well-documented* code [124]. The authors highlight the requirement for views of the program as code or comment dominated, even with the same program and user combination. They believe that the WEB approach is disadvantageous for three reasons:

1. Refinement addition tends to encourage global structure over nested block structure
2. Rearrangement of program fragments creates an inconsistency between the user's view of a program and its executable semantic structure
3. Re-arrangement is not well suited to exploiting hierarchical abstraction features proposed

The approach which they took was to add an optional *comment pane* into each block window. The percentage of the space used by the comment pane can be adjusted by the user, to cope with users unfamiliar with the code as well as experts. Block comments are displayed in this pane, using the normal text-editing paradigms appropriate to text areas. Re-arrangement of the code into modules was rejected for the reasons provided, and the granularity of the block comments was not adjustable. Multilingual document editing was realised using a generalisation of this method of associating text zones with blocks. The document is seen as a hierarchy of document contexts each consisting of a sequence of zones in different languages, including text.

Cordes and Brown's review of the literate programming paradigm [30] suggests some modifications to bring it up to date. Four additions are proposed; a multilevel table of contents, a graphical user interface, program debugging tools and an enhanced index. The authors have implemented prototype versions of the second and third of these additions, and are working on the other two.

They also believe that restrictions should be made on the structure of the literate program and on the size of the command set for driving the literate programming tool. Each module in the program should function as a logical single entity, within the syntax rules for the language. This assists the reader in understanding module functionality and tracking the program scope. The second restriction removes some of the individual customisability of the output listing, but a template-based documentation generation routine is envisaged which could tailor the output for particular house styles. These restrictions would be easily implementable in a structure-based literate programming editor.

A collection of Knuth's papers on structured programming and literate programming has been published recently [69]. While there is very little new material in this book, it does bring together Knuth's thoughts on the topic of literate program, with some of the earlier papers which led to the development of literate programming, and some examples of what Knuth's literate programming has achieved.

2.1.10 Bibliographies

Two useful bibliographies of literate programming material are by Smith and Samadzadeh [106], which contains a reasonably comprehensive list up to the start of 1991, and Thomas and Oman [113], which contains references which are more relevant to programming style and programmer comprehension studies.

2.2 Syntax-directed Editing

The Emily system [51] was one of the first syntax-directed editors. It was implemented on an IBM 360 mainframe computer equipped with IBM 2250 display terminals, and accepted input from a light pen and keyboard. Emily presented the user with a view of the abstract syntax tree of a language, displayed according to a set of concrete syntax rules. The system allowed replacement of non-terminal symbols of the grammar by instances of their productions, selected from a list of possible choices. Non-terminals were highlighted on the display to indicate that they needed to be expanded. Expansions of non-terminal symbols could be contracted and displayed in abbreviations called *holophrasts*, which used the name of the non-terminal symbol and the first few characters of its expansion as an abbreviation for the hidden fragment. This abbreviation could be used in commands that referred to the fragment of code. The Emily system was table-driven; abstract and concrete syntax tables for PL/I, GEDANKEN, a thesis outlining language and a language for syntax definition were created.

The methods used in Emily re-appear in most of the syntax-directed editors developed since, especially the use of an abstract syntax tree for the internal representation of a program, template expansion at non-terminal symbols, and computing the display representation on demand by concrete syntax rules.

The Cornell Program Synthesizer [109, 110] was a template based syntax-directed editor for the PL/CS programming language (an instructional subset of PL/I). The program was developed top-down by inserting templates at appropriate placeholders in the parse tree. Textual input could also be made at placeholders, which caused the text to be parsed and inserted in the parse tree if it was correct. Syntactic correctness of the edited program was guaranteed because the templates were predefined and could not be altered or put in inappropriate places, and the parser checked textual phrases for correctness before accepting them. The cursor could only be moved by structural units, although phrases could be edited textually and re-parsed as if they had been inserted at a placeholder. Template to template transformations were provided to allow mutation of templates into other forms without complicated sequences of cut and paste operations.

Code was generated for each structural unit as it was accepted by the editor, so the program was always in a runnable state. The static semantic checking preformed by the incremental code generator was used to highlight incorrect phrases until they were corrected. The program could be run at any time, and the display cursor would follow the flow of control. If an error occurred, the cursor would stop at the position of the error and an error message would be displayed. This allowed incomplete programs to be executed, which would stop when a placeholder was encountered.

Templates were the computational units in the editor; at runtime, they could be used for single-stepping. The execution rate could be slowed down so that the flow of control could be seen more clearly, by setting the time unit that each step should take. A special sort of template called a *comment template* was provided, which contained a comment and a sequence of statements. The display of this template could be collapsed and expanded by pressing a key, hiding all of the statements under an ellipsis sign. Such collapsed templates were treated as one unit by the execution system; the operations in them would all be performed in one time unit if possible. The comment template was provided as an incentive to use comments, to help comprehension of the program, and simplify the information viewed when debugging. The Cornell Program Synthesizer led on to the development of the Cornell Synthesizer Generator.

IPE (incremental programming environment) was part of the Gandalf software development environment project at Carnegie-Mellon University [78]. The user interface to IPE was through a syntax-directed program editor, which was augmented by incremental program compilation and execution. The display was created by unparsing the abstract syntax tree according to an unparsing scheme; an interesting point is that alternate unparsing schemes were supported, to allow viewing the program at different abstraction levels, *e.g.*, lists of modules, module specifications, and module implementations. Textual manipulation of the unparsed program representation was not supported. A parser and lexical analyser were not provided because all of the IPE tools used the abstract syntax tree representation of the program; this restricted the programs that IPE could handle to programs constructed with IPE itself.

The GNOME family of editors [46] were created to channel the experience gained in the Gandalf project into a practical novice programming environment. Four structure editors with common interfaces were created: they were a family tree editor, a Karel editor, a Pascal editor, and a FORTRAN editor. The family tree editor was used to familiarise students with structure editing concepts, such as walking trees, hierarchical structure, nodes, subtrees, and tree operations. The Karel editor included complete static semantic checking of the language, and was linked to a Karel interpreter so that the students could move between program construction and execution without leaving the editor. The Pascal editor contained semantic checking for the most common errors that the students made, but did not contain full semantic checking. A Pascal interpreter was linked to the editor to allow execution of the programs without leaving the editor.

The GNOME editors were built using the AloeGen structure editor generator created by the Gandalf project. The AloeGen generator was a structure editor which generated an editor from a BNF-like description of abstract syntax of the target language. Unparsing specifications could be provided to define the displayed representation of the abstract syntax tree, and action routines could be specified which were performed whenever the pieces of the syntax tree with which they were associated were changed. The action routine mechanism was used to implement static semantic checking and incremental compilation in some editors. A set of extended commands could also be supplied by the implementor, to perform actions outside of the normal editing task. The GNOME editors have been used to teach programming to an undergraduate population of about 1500 at Carnegie-Mellon University.

The MENTOR system [35, 36] was developed to manipulate structured data represented as abstract syntax trees. The abstract syntax trees used by MENTOR were not parse trees, although they could be obtained from by collapsing and normalising parse trees. Some of the important differences were:

- Lists were represented as one list node, rather than as a binary tree.
- The reserved words of the language grammar appear as node labels rather than leaves of the tree.
- Non-terminals of the grammar do not generate nodes. The abstract syntax tree is structured as a sorted algebra, which is defined by a set of *sorts*, and a set of *operators* with sorted operands. Some non-terminals of the grammar correspond to sorts (which are represented as nodes in the abstract syntax tree), others do not appear at all. For example, an identifier may occur directly as an expression, the intermediate levels of parsing such as factors, terms, *etc.*, being collapsed. Every node of the abstract syntax tree makes a visible mark on the display because of this, so the user can directly relate the display to the underlying tree structure.
- Parentheses are not part of the structure, but were generated by the unparser if the context required them.

The user performed operations on the abstract syntax trees by invoking procedures written in the MENTOL tree manipulation language. MENTOL had variables called *markers* which could be locations in trees, and values which could be locations or abstract syntax trees. Operations such as tree copying, sub-tree deletion, tree traversal and rewriting, and pattern matching could be written in MENTOL. MENTOL itself was defined by an abstract syntax, and advanced users could use MENTOR to edit and create new MENTOL procedures to perform program transformations.

Attributes could be attached to nodes of the abstract syntax tree, to define comments, program documentation, program assertions, cross-references, *etc.* The values of these attributes were abstract syntax trees in their own language, and could be accessed by MENTOL procedures. Using attributes to attach this information meant that the editor could be extended to perform different tasks, such as program verification, with an appropriate set of attributes and MENTOL procedures. However, these extensions would not interfere with the procedures for editing the abstract syntax tree. Donzeau-Gouge *et. al.* [36] compare this technique to Knuth's WEB, in which a Pascal program and its documentation are intertwined. MENTOR is not comparable as a literate programming system, because the program's abstract syntax tree defines the primary view in MENTOR, rather than the documentation which was represented by attributes. The MENTOR created program cannot be re-ordered for presentation as a WEB program can, failing one of the criterion for literate programming presented in section 2.1.2.

A structure editor for Pascal was built using MENTOR, primarily to help create the MENTOR system itself (it was written in Pascal). The syntax of the Pascal program was guaranteed to be correct by the editor because of the underlying abstract tree representation of the program. Procedures were written to perform some semantic checking and optimisation of the program, but some peculiarities of the Pascal language prevented a clean separation of the various checking stages. Donzeau-Gouge *et. al.* [35] suggest that "The conclusion we can draw from this state of affairs is that no really satisfactory programming environment will exist for ugly languages." MENTOR could read and parse normal text files, and write them out either as unparsed text files, or as tree files which could be re-loaded without parsing.

Poe (the Pascal Oriented Editor) [44] took a different approach to syntax-directed editing than most of the other systems described. Instead of constraining cursor movement and template expansion to structural units, Poe contained an interactive parser which used an automatic repair algorithm to ensure that the program stayed syntactically correct. The error repair algorithm determined where to insert tokens to ensure that the textual input remained correct, using a least cost calculation to choose between alternative expansions. Required and optional *prompts* were put in the expansions inserted by the error repair algorithm to indicate where further expansions could be made. For example, if the user typed the token “if” at a statement prompt, the required prompt for an expression, the token “then”, and an optional statement and an else-clause prompt would be inserted. Fully-formed template insertion was not provided, but the possible expansions for prompts could be requested, with repeated requests cycling through the available expansions for a prompt. An incremental parser was used to drive the error-repair mechanism; backtracking was supported in the parser so that deletions did not require the whole program to be parsed again. In the cases that the error-repair mechanism expanded the prompt incorrectly, and undo facility could be used to correct its mistakes.

The parse tree generated by the incremental parser was attributed, and an incrementally-evaluated attribute grammar was used to detect static semantic errors in the program. These errors were highlighted in the program display until they were corrected. Poe input and output the program as text, which had the advantage that other programming tools could operate on the program, but the disadvantage that there was a relatively large start-up time as the program was parsed. The textual output was used to transparently invoke the system's standard compiler when a request was made to execute a complete program. Unlike other systems including incremental compilation, Poe could only request execution of complete programs.

Much of Poe was table driven, and a language-based editor generator based on the features available in Poe was also developed. The Poegen editor generator automatically created the incremental parser and error-correction tables for Poe editors from a context-free grammar augmented with attribute evaluation rules.

The UQ2 program editor [124] was created with the intention of producing programs which were well-formed (in a syntactic and semantic sense), well-formatted (to improve readability), and well-documented (to make design choices leading to the program apparent). The text input and incremental parsing approach used in Poe was chosen for program input, allowing touch typists to operate at their normal speed, but less experienced typists to benefit from the automatic template generation. The incremental parsing method ensured syntactic correctness, and with the use of incremental semantic analysis to detect and display semantic errors, well-formed programs could be produced.

Well-formatted programs were produced by using incremental adaptive pretty-printing, which reduced the space taken by trivial occurrences of potentially large constructs to a single line, while allowing large constructs to adopt the minimum multi-line format consistent with their usage. Blocks could be suppressed or zoomed in by user commands to give clearer views of code. A measure of structural distance was used to automatically suppress code to provide a clear view of the overall structure of the code.

Two comment conventions were supported by the UQ2 editor in order to assist creation of well-documented programs. Block comments paralleled the code block nesting, and embedded comments could be used for local clarification within blocks. The editor presented each block with a comment pane and a code pane. The comment pane provided a simple text editing and scrolling interface which could be suppressed by experts who knew the code. A facility was provided to print the program with in the top-down depth-first order of blocks, with the block comments typeset by a formatting program such as TEX or *troff*. The re-arrangement of the program that WEB performs was not attempted by the authors, because they felt that it discouraged use of block structure, and made the programmer's view of the executable semantic structure inconsistent.

The Syned [56] language-based editor was also based around parsing of text input into an internal attributed abstract syntax tree, but lacked the error-correcting parser which kept programs edited by Poe syntactically correct. Textual changes were submitted as transactions which could be translated into deletions and additions to the internal abstract syntax tree representation.

The PECAN [95] program development system used a syntax-directed editor similar to the Cornell Program Synthesizer as one view of the program. The editor provided template expansion and textual editing and re-parsing of constructs, and maintained an internal representation of the program as an abstract syntax tree. The abstract syntax tree was used to compute the other views of the program available in PECAN; these included:

Nassi-Schneidermann view. This was a graphical view a Nassi-Schneidermann flow chart representing the program. It could be edited, with the changes being reflected in the abstract syntax tree shown by the syntax-directed editor.

Declaration view. Declarations could be viewed by pointing at the name of a variable or type in the program and selecting the declaration view. These declarations could then be modified to change their names, types, or class, or even moved to a different scope.

Symbol table view. All of the scopes and symbols in the abstract syntax tree were incrementally added to the symbol table view. The symbol's class and type were displayed for each scope. This view was read-only.

Data type view. While the user was editing variables or type definitions, the data type was displayed in a read-only data type view. Recursive definitions were only expanded to one level, but the user could choose to expand further definitions interactively.

Expression view. The expression view displayed a graphical representation of expression parse trees as expressions were edited.

Flow view. The incremental compiler used in PECAN built and maintained graphs describing the flow of control through the abstract syntax tree. Nodes of the graph represented expression evaluations, conditional branches, variable allocations, *etc.*

Execution views. There were three views of program execution in PECAN; control, program, and data. The control view displayed the program's current state, and input and output to and from the program. Debugging breakpoints and single-stepping were supported from the control view. The program view highlighted the statement currently being executed, and the data view showed the program stack as the program executed.

PSG, the Programming System Generator [11], incorporated a hybrid editor which allowed structure-oriented as well as text editing. The PSG editor different from many of the other syntax-directed editors by not enforcing a top-down expansion of the program. Unconnected program fragments could be developed separately, and combined when the programmer desired. In PSG, the syntax and static semantics of the target language were specified by an attribute grammar, and a description of the dynamic semantics could be given in denotational semantics. The attribute grammar was used to perform type inference on the unconnected program fragments, allowing static semantic checking of program fragments which had not been placed in context.

The Incremental Programming Support Environment (IPSEN) [38, 39] used *graph grammars* to specify the data structures which could be manipulated. Graph grammars use graph rewriting rules to specify how graphs can be derived from ancestor graphs. The graphs in IPSEN were used to describe the structure of the document or programming language being edited, and attributes of these graphs represented the contents of the nodes of the graph. In order to achieve reasonable performance without too much complexity, attributes were provided which could contain quite complex structures such as whole paragraphs of text, and specialised editing operations on these attributes were provided. The graph-rewriting rules were used to drive a template-based editing front end.

The Synthesizer Generator [96–98] is probably the best known syntax-directed editing system. It is the successor to the Cornell Program Synthesizer. The Synthesizer Generator is a syntax-directed editor generator, rather than an editor itself, but editors have been implemented with it for the computer languages C, FORTRAN, Pascal, and SSL, as well as systems for formal logics, balanced chemical equations, picture specification languages, lambda calculus interpreters, outline specifications, and program verification. The editor is described in the Synthesizer Specification Language (SSL), which defines the syntax and semantics of the language in a context free grammar augmented by attribute equations. A concrete input syntax can be defined, and multiple unparsing specifications can be provided to create different views of the edited program. Applicative functions can be defined for use in attribute equations, and transformations can be defined to restructure the abstract syntax tree. The attribute equations are evaluated by an incremental attribute evaluator, providing instant feedback of semantic errors or other attribute-based computations. Editors generated by the Synthesizer Generator are template-based, but text can be input at placeholders if a concrete syntax is provided for the corresponding abstract syntax. Intermediate layers of the parse tree can be made transparent to the user by providing resting places for the selection at only the desired points in the abstract syntax tree. Files can be read and written in text, structured, or attributed structured form.

Most of the syntax-directed editors described already have been *template construction* systems; *i.e.*, the program was developed top-down by selecting expansions from a list of possible templates at each point. The syntax of the programming language has been enforced either only allowing construction by template expansion, or by refusing to accept text which cannot be parsed as an appropriate construct for an insertion.

The Z editor [126] was a text editor that provided support for program construction. Wood claimed that Z could do 95 % of what could be done in a program editor without increasing its complexity significantly. Z provided automatic indentation of program constructs, and the indentation could also be used to select, suppress, and move over program constructs. Z could automatically balance expressions and move over balanced expressions. Compilation of programs could be started asynchronously, with any error messages displayed at the bottom of the screen. The cursor could be automatically moved to the site of the each error in turn. Z did not perform any form of syntactic or semantic checking of programs, so immediate feedback of errors was not possible.

While Z had good facilities for communicating with other programs, each program which might be run on the source code (compilers, program verifiers, pretty-printers, *etc.*) had to parse the program itself, slowing down feedback and duplicating effort. Neither multiple interlinked views provided by PECAN, nor the comment panes of UQ2, the incremental compilation of the Cornell Program Synthesizer, or the error-correcting input of Poe could be performed in Z. While a programmer is editing a program, there are plenty of pauses which can be used to do useful work in translating the program; this is what most syntax-directed editors rely on to provide a tool which provides instant feedback about syntax, semantics and structure of the program.

3 Objective

3.1 Introduction

Literate programming systems have been around for some time, but have not gained widespread acceptance as programming tools. There are several reasons why this may not have happened:

- More effort is required of the programmer when writing literate programs than using conventional techniques. For example, with Knuth's WEB, the syntax of the macro language must be learned, the formatting language (TEX) must be learned, documentation sections must be created, named, and referenced properly. Trevorrow's criticism of WEB [114] noted that the user has the possibility of making errors in three languages at once.

- In the absence of any better tools, a normal text editor must be used to create WEB programs — this is a potential strength, as well as a weakness, of WEB — the program source is easily transportable between machines, and can be edited in an environment which the user is familiar with, at the expense of extra effort in other areas.

- Programs written with literate programming systems are less portable than conventional programs, because the literate programming system must be available to process the program. There have been attempts to create literate programming systems which enclose the documentation inside normal comments, but these systems fail one of the fundamental tests of literate-ness, in that they do not allow a flexible order of elaboration.

The first two points are what the development of the literate programming editor in this thesis are targeted at; creating an incentive to develop literate programs, by reducing the effort required of the programmer. An ideal system would be one in which it is easier to write literate programs than it is for the programmer to use conventional tools. A sufficiently flexible system should be configurable enough to behave similarly to the programmer's favourite editor.

The third point must necessarily stay a hinderance to the acceptance of literate programming, because any method of indicating the documentation structure of a program which leaves the program in its original form fails either the criterion of verisimilitude, and can be subverted by directly editing the program, or fails the criterion of flexible elaboration, because the program cannot be presented in any order desired. The only way in which portability can be provided is to allow exporting the final program from the literate programming system to conventional programming systems (as is done by Ramsey's noweb). Files exported in this way should be treated as read-only, and should not be modified.

In addition to these reasons for lack of acceptance of literate programming, there are problems with the implementation of many literate programming tools:

- The macro-processing mode of operation of WEB-like tools may allow subtle errors to creep in, because substitutions are textual, and not structural.

- Debugging literate programs may be more difficult than conventional programs, because the parts of the program presented to the compiler were presented in a totally different order to the programmer. Statements which are next to each other on the pre-processed output may have come from completely different sections of the original document, making it difficult to relate the sources of errors.

- Many literate programming systems are missing important parts of the paradigm which makes literate programming effective, such as indexing and module cross-referencing. These systems require yet more effort from the programmer to achieve the same results, reducing the incentive to use literate programming. The productivity improvements possible from having these tools available while developing the program will not be realised.

The rest of this chapter will look at these problems in more detail and show how syntax-directed editing could be used to solve them.

3.2 Control, formatting and programming languages

Knuth's WEB system used three languages; the WEB control language, the TEX formatting language, and the Pascal programming language. A literate programmer using WEB should reasonably be expected to know Pascal, but the addition of TEX and WEB control sequences complicates matters somewhat. There are three escape characters to be borne in mind (@ introduces a WEB command, \ starts a TEX command, and | is used to delimit Pascal text in documentation sections). The WEB command language contains 27 commands, several of which make esoteric alterations to the spacing of the Pascal code or control the automatic formatting performed by the weave processor, and a parameterised macro definition and expansion capability. The TEX formatting language adds even more complexity, and requires years to master fully — even then it is possible to find examples to which TEX wizards will say "I didn't know it would/could do that!".

Syntax-directed editing could be used to simplify this mess. Structural information such as the contents of documentation and code parts of refinement sections could be indicated visually, using space and indentation, or even different typefaces and colour to offset the code parts. Documentation could be presented as a sequence of paragraphs containing text. The text itself should have a simple structure; the characters that the programmer types should not have hidden surprises, like escape characters for formatters or control languages, but should indicate what will appear in the final document. This is not the same as WYSIWYG (what you see is what you get) presentation, because there does not need to be any attempt to present the final appearance of the document on the screen. There is however a correspondence between the characters typed in the documentation and the characters appearing in the printed documentation.

Simple markup abstractions for common constructions (such as emphasis, lists, *etc.*) could be defined, which would be mapped onto code in the formatting language when the program is output for formatting. These markup abstractions could again be indicated visually, and templates for their use provided. More advanced control of the formatting could be provided by providing structures which pass the text contained within them through to the formatter.

Visual differentiation can be used to indicate different structural entities, using colour, different typefaces, and spacing.

3.3 Textual decomposition of programs

There are problems related to the textual decomposition of the program into modules which is facilitated by WEB and its progeny. In WEB, module references may be inserted at any point in the program, and a purely textual substitution is performed before the program code is written out for compilation. Modules therefore do not necessarily contain syntactically complete or correct fragments of code. In practice, module references are nearly always inserted at certain well-defined places (such as statements, expressions, declarations, and case branches). This is probably because it is simpler to comprehend the function of a module if the context in which it is used is easily defined and understood.

Textual decomposition of the program into modules can easily introduce errors which are difficult to detect. For example, consider a WEB fragment which contains a module reference:

```
if test > value then
  @<Perform operation on value threshold@>
```

and the definition of the module:

```
@<Perform operation on value threshold@>=
  write('Error: value is too high')
```

This fragment of code contains the hidden assumption that the module "Perform operation on value threshold" contains just one statement. (In Pascal, the branches of an `if...then...else` statement contain a single statement each. If more than one statement is needed, a compound statement is created using the `begin...end` construct.) If a programmer altered the module definition, without careful consideration of the context in which it was used, the definition might end up like this:

```
@<Perform operation on value threshold@>=
  write('Error: value is too high'); errorflag := true
```

In this case, only the first statement in this module will be performed when the condition is true; the other statement will always be performed, possibly leading to erroneous results. The program is syntactically correct, but the structure imparted by the division into modules does not actually match the syntactic structure.

It is easy to reply that the context in which the module is used should always be considered carefully. While this is in general true, the semantics of the code in which a module is used may be altered by the module definition, and so the context in which this code is used should be carefully considered, and so on. This interdependency of modules in WEB code points to the need for extra structure beyond plain textual substitution.

A more complicated example is the case in which the first piece of code contains an `else` branch:

```
if test > value then
```

```
    @<Perform operation on value threshold@>
  else begin
    @<Normal operation for value@>
  end
```

In this case, a problem arises if the module "Perform operation on value threshold" is modified to act conditionally:

```
  @<Perform operation on value threshold@>=
    if verbose then write('Error: value is too high')
```

In this case the module definition is still a single statement, but after textual substitution the `else` clause of the code which uses it becomes attached to the `if` statement within the module.

Both of the examples provided here are closely bound to the semantics of the programming language in use (in these cases, Pascal). Any programming language will have its own share of potential ambiguities; several authors have said that the full power of literate programming will only be realised when it is integrated into a programming language designed for the purpose, rather than retro-fitted to languages designed for other purposes. Unfortunately, this is unlikely to be a popular option, because of the delays in creating tools, compilers, and support environments for new languages, and the inertia in getting users to program in a new language.

Syntax-directed editing could completely prevent problems caused by textual substitution. In the first example above, a syntax-directed editor would recognise that the definition of the module must contain a single statement, and complain when a statement list was used instead. The second example is likely to be recognised by a syntax-directed editor as a case of the "dangling `else`" problem, and appropriate preventive action taken or warning issued. The price to pay for introducing syntax-directed editing is a slight loss of flexibility in positioning of module references. None of the literate programs which were read and reviewed when performing the literature survey in chapter 2 or designing the implementations in chapter 4 had module references which crossed syntactic boundaries, so the loss in flexibility is likely to be negligible if the reference positioning is designed carefully.

3.4 Debugging

In the WEB system, Knuth deliberately obscured the output of the `tangle` processor, to prevent programmers from altering the Pascal code directly. Statements were put on the same lines, keywords and identifiers were put in uppercase, comments were removed, and line breaks were placed in unusual places. The only concession to debugging was to insert comments indicating where the starts and ends of modules were. Unfortunately, this output made the use of source-level debugging tools (which were not common when WEB was created) very difficult, because the code that the debugger shows bears very little resemblance to what the programmer recognises. Even if bugs are discovered with a debugger, finding the point at which a fix should be made back in the original WEB source requires more effort from the programmer.

This problem has been tackled by some systems, such as Levy's CWEB, by introducing compiler directives which indicate the file and line in the original source code that each module comes from. Source-level debuggers can then use this information to display the original CWEB file, rather than the intermediate C program. Levy's CWEB uses the C preprocessor's `#line` directive, which is a standard feature in implementations of the C language. However, there is no standard equivalent of `#line` for many other languages such as Pascal, so this technique cannot be used in these cases.

A syntax-directed editor which has knowledge of the semantics of the programming language could prevent many of the errors which the compiler detects. Syntactic errors would be prevented because the program would always be syntactically correct, and many semantic errors could be detected and indicated in the editor's display, preventing the erroneous code from being sent to the compiler in the first place. Logical errors in the program design or implementation are still possible, but incremental code generation and execution could be incorporated to allow single-stepping through the literate program to facilitate catching these errors.

3.5 Publishing Tools

Knuth's WEB system automatically generates a table of contents, an index of the identifiers appearing in the program, and cross references for each module indicating the module in which it is used and any other modules which augment the module's definition. Unfortunately, this information is nearly always out of date, as printed copy of the program tends to be made rarely because of the time and expense involved, and almost any change to the program will invalidate this information.

Brown [23] showed that having this auxiliary information available made navigating around the literate program much easier. A syntax-directed editor cogniscent of the structure of the documentation could generate these lists interactively, and either provide separate windows displaying the contents and index, or display them at the start and end of the edited document. The module cross-references could be displayed within the modules. It may also be possible to provide navigation links between sections, so that the programmer could easily find the next module which augments the current module, the module which includes the current module, or the next module which uses a selected identifier. The editor could also automatically number sections, so that moving or removing sections would keep the section ordering, table of contents, and module lists up to date.

3.6 Additional benefits

The main reasons for implementing a syntax-directed literate program editor have been explained in the previous sections. There are some more potential benefits that such an editor could have:

Cross-referencing. There are implicit and explicit cross-references between the code of a WEB program and its documentation. Documentation may describe pre- and post-conditions for code chunks, using variable and procedure names from the code. It may be possible to link identifiers appearing in the document with identifiers appearing in the code, so that a change to the identifier in the code would either change the corresponding identifiers in the documentation or give a warning that the identifier in the documentation is not defined.

Auxiliary information. The notion of *verisimilitude* which literate programming introduced to indicate that the documentation and code came from the same source file breaks down when auxiliary information such as tables, graphs, and example output are introduced into the documentation. For instance:

- Command syntax for a program may be specified by a grammar, which appears as a pretty-printed grammar or railroad diagram in the documentation, with a corresponding parse table and parsing procedures in the code. A syntax-directed editor which has knowledge of the structure of the grammar could generate the diagrams automatically. This example would probably be better supported by invocation of an auxiliary program which generates both code and documentation from a single file.

- Output of a function or the entire program appears in the documentation, with the example or test input which generated it. A syntax-directed editor which supports incremental code generation or invocation of external programs might be able to automatically re-generate the documentation when the corresponding code or test input is changed.

The concept of test interfaces in the example above is used in Reenskaug and Skaar's SmallTalk literate programming system [94].

3.7 Summary

The problems associated with WEB and similar tools and systems can be summarised thus:

1. The programmer is required to learn more syntax than is relevant to the task, *i.e.*, designing, writing, and documenting a program.
2. The transformation of WEB programs into compilable source code operates by purely textual substitution; subtle errors may be present in WEB programs which will not be caught by this process.
3. Programmers may not be able to debug the same source code that they wrote, and are left to struggle with the ugly-printed intermediate files.
4. Navigation information is only contained in the printed copies of the source code, which quickly become obsolete.

A syntax-directed editor with knowledge of the structure and semantics of the programming language and the structure of the documentation could help solve all of these problems.

3.8 Strategy

Documentation may be required for a program for several different purposes:

User manual. A tutorial to help new users of the program.

Reference manual. For experienced users to find concise information in.

Maintenance commentary. To ease porting to other architectures and functional changes.

Pedagogic commentary. A commentary on interesting algorithms and data structures to allow them to be re-used in other programs.

The literate programming paradigm is most appropriate for production of the latter two forms of documentation, because of their intimate relationship with the program code. There is still a need for consistency with the former two types of documentation, but this may be better addressed by other means.

The objective of the next stage of work was to develop a syntax-directed editor for literate programs based on a sufficiently powerful system that it could be used to derive and check information about the semantics of the programming language chosen.

The *attribute grammar* description of programming languages [63, 64] is a well-known and powerful model for representing programming languages; the syntax of the programming language is specified by a context-free grammar, and the semantics can be specified by attribute evaluation rules. There are several syntax-directed editing systems based on attribute grammar evaluation, including the Cornell Synthesizer Generator [97], PSG [11], and Poe [44]. To use attribute grammars to represent literate programs, the context free grammar would have to be extended to describe the structure of the entire literate program with the documentation included.

The initial phase of this work was to examine whether the Cornell Synthesizer Generator was a suitable framework on which to base the literate program editor. The Cornell Synthesizer Generator was chosen as a base on which to develop the literate programming editor for several reasons:

Availability. At the time when it was chosen, the Synthesizer Generator had a very reasonable academic-use license, and support was provided for it.

Completeness. The Synthesizer Generator has been used for several complete program editors before, for different styles of programming language. It was thought that it would be the most capable tool for implementing a program editor.

Ease of specification. The entire editor, including input lexical units, abstract syntax, attribute equations and display representations are written in the

SSL language. This is a functional language with a reasonable set of primitives for data manipulation and flow control.

External linkage. The SSL language allows linking attribute evaluator functions written in C into editor specifications, to provide special capabilities which could not be written (or could not be written efficiently) otherwise.

The plan for the implementation work was as follows:

Stage 1. The first step was to create literate programs editor for "toy" programming languages investigate how to represent literate programs with attribute grammars.

Stage 2. An editor for a real programming language would then be developed, initially to demonstrate how the code and documentation could be presented so that no knowledge of the control language or structure was necessary.

Stage 3. The editor would then be extended to include knowledge about the semantics of the programming language, demonstrating how semantic as well as syntactic errors could be detected and displayed in a literate program.

Stage 4. Finally, publishing tools such as a table of contents, an index of identifiers, and module cross-references would be added. Some of the additional benefits made possible by using syntax-directed editing might then be investigated if time permitted.

4 Implementation Studies

4.1 SSL, the Synthesizer Specification Language

Editors created with the Synthesizer Generator are written in the Synthesizer Specification Language (SSL). An SSL specification consists of a list of declarations, which describe the abstract syntax, concrete syntax, and operations of the target language. The core of a specification is the abstract syntax, described by a set of grammar rules. These grammar rules only contain nonterminal symbols; *entry declarations* are used to define how subsets of the *concrete syntax* map onto the abstract syntax.

The grammar is also a type-definition mechanism in which the nonterminal symbols are names for types which denote sets of values. These sets of values are the set of derivation trees (or *terms*) derived from a given nonterminal symbol, and are known as *phylum*. Each production in the grammar derives terms that can be treated as records; the alternatives of a production derive different record variants. Terms are used as abstract representations of objects to be edited and also as computational values. Each production in the grammar has a name, known as an *operator*, that can be used in computational expressions, in different contexts as a record constructor and as a selector that discriminates between variants. Productions, nonterminal symbols, and operator names are defined simultaneously in *phylum declarations*. A single *root declaration* specifies the root nonterminal symbol of the grammar.

Calculations on terms can be specified by attribution rules. *Attribute de-*

clarations associate attributes with nonterminal symbols and productions, and *equation declarations* define the values of these attributes in terms of other attributes. Expressions for evaluating attributes are written in a strictly functional style; there are no side effects. *Function declarations* can be used to create abstractions of common operations. The Synthesizer Generator automatically updates all of the attributes which are affected after each editing operation.

The editing interface to Synthesizer Generator editors is a set of windows displaying some of the *buffers* which have been loaded from files. Each window displays one *view* of one buffer; several different windows may display alternative views of the same buffer, in which case all of the windows will be updated as each editing operation occurs. Each buffer has a *selection*, which is the subterm or sublist of current interest. The selection is highlighted in each window in which it is visible. Each editing transaction replaces the selected subterm or sublist of the buffer with another. These replacements may be specified by *transformations declarations*, which specify how to restructure an object when the selection matches a given structural pattern, or by text editing. Entry declarations specify which portions of the concrete input syntax are recognised with the current selection, and how the parse tree of a textual input is attributed to create the corresponding abstract syntax. Each of the terms of the abstract syntax has an *unparsing declaration*, which defines how the term is displayed on the screen in each of the views available, and whether it may be selected. Unparsing declarations can control the level of indentation. [illegible] breaks, and whether the subterms and attributes of productions ar[illegible] the display. Different fonts and font characteristics can be invok[illegible] [illegible]e unparsing declarations through *style declarations*. There are co[illegible]nds to traverse the abstract syntax tree, which may be bound to [illegible] or key-sequences. Locating devices such as mice may also be us[illegible] select terms of interest in the buffer. The default key bindings for [illegible]nthesizer Generator editors are reminiscent of Gnu-Emacs, but may be changed easily.

4.2 Design issues for literate program editors

There is a fundamental design decision to be made when creating a literate program editor with the Synthesizer Generator, which is the choice of structuring the editor with the program structure as the primary structure or the documentation structure as the primary structure. A literate program can be viewed as two intertwined structures, one of which defines the program, and one of which defines the documentation (the documentation actually *includes* the program as well). The editors created by the Synthesizer Generator have only one root to their abstract syntax, so the secondary structure must be incorporated by interleaving syntax declarations at the appropriate points and using attribute evaluations to provide error and type checking.

Using the program structure as the primary structure for an editor has some advantages:

- *The (context-free) syntax of the programming language can be enforced by the editor.* This does not imply that any program created with such an editor will be *correct*; there may be logical errors in the program (such as infinite loops), semantic errors (such as using undeclared variables), or the program may be incomplete. However, syntax errors will be eliminated.

- *Conversion of non-literate programs to the literate programming idiom is simplified.* If a complete concrete input syntax is defined for the language, existing programs can be read in and then documented. This is not a very great advantage, because the literate programming paradigm works best when programs are written from scratch, with the exposition of the underlying algorithms foremost in the author's thoughts.

There are corresponding disadvantages when using the program structure as the primary structure:

- *Displaying the literate program in a sensible order is difficult.* The order in which documented sections should be displayed is not necessarily the same as the order in which they appear in the program, and re-arranging the display order may be difficult. Compressing the display so that pieces of code which are "abstracted away" in other sections are displayed as section references may be difficult. These two problems may require that the normal display mechanism of the generated editor is by-passed, and that the representation for every node in the abstract syntax tree is defined as an attribute of that node. (If there were some mechanism to alter the unparsing mode of the editor conditionally on where the node is in relation to the current selection, this would not be so.) This would not only be inefficient, but there may probably also be problems relating the current selection to the original abstract syntax node.

In addition, sections of documentation which have no corresponding code section may be difficult to incorporate into the structure.

Using the documentation structure as the primary structure for an editor has some advantages:

- *Displaying the program is simple.* Arranging the order of display, and displaying abstracted parts of code is simple, because the literate program's structure follows the order of explanation.

The corresponding disadvantages when using the documentation structure as the primary structure are:

- *The syntax of the programming language cannot be enforced by the editor.* This is possibly not such a great disadvantage, because the editor cannot *enforce* the semantics of the language either. The usual recourse when a semantic error is detected is to insert a warning message in the display, and this method can be followed for the cases in which the syntax cannot be directly enforced. The warning messages can be generated by passing attributes through the documentation structure which define the syntax of acceptable code sections.

- *The concrete input syntax may be non-context free.* Parsing of the program may be more difficult in some cases, because the left-context may be impossible to determine. This may prevent writing a concrete input syntax for the entire literate program (*i.e.*, documentation and program), but parsing rules which use the context of the (hidden) abstract syntax may still be written to allow unambiguous parsing of conflicting rules.

Version 3.3 of the Synthesizer Generator was used for these studies.

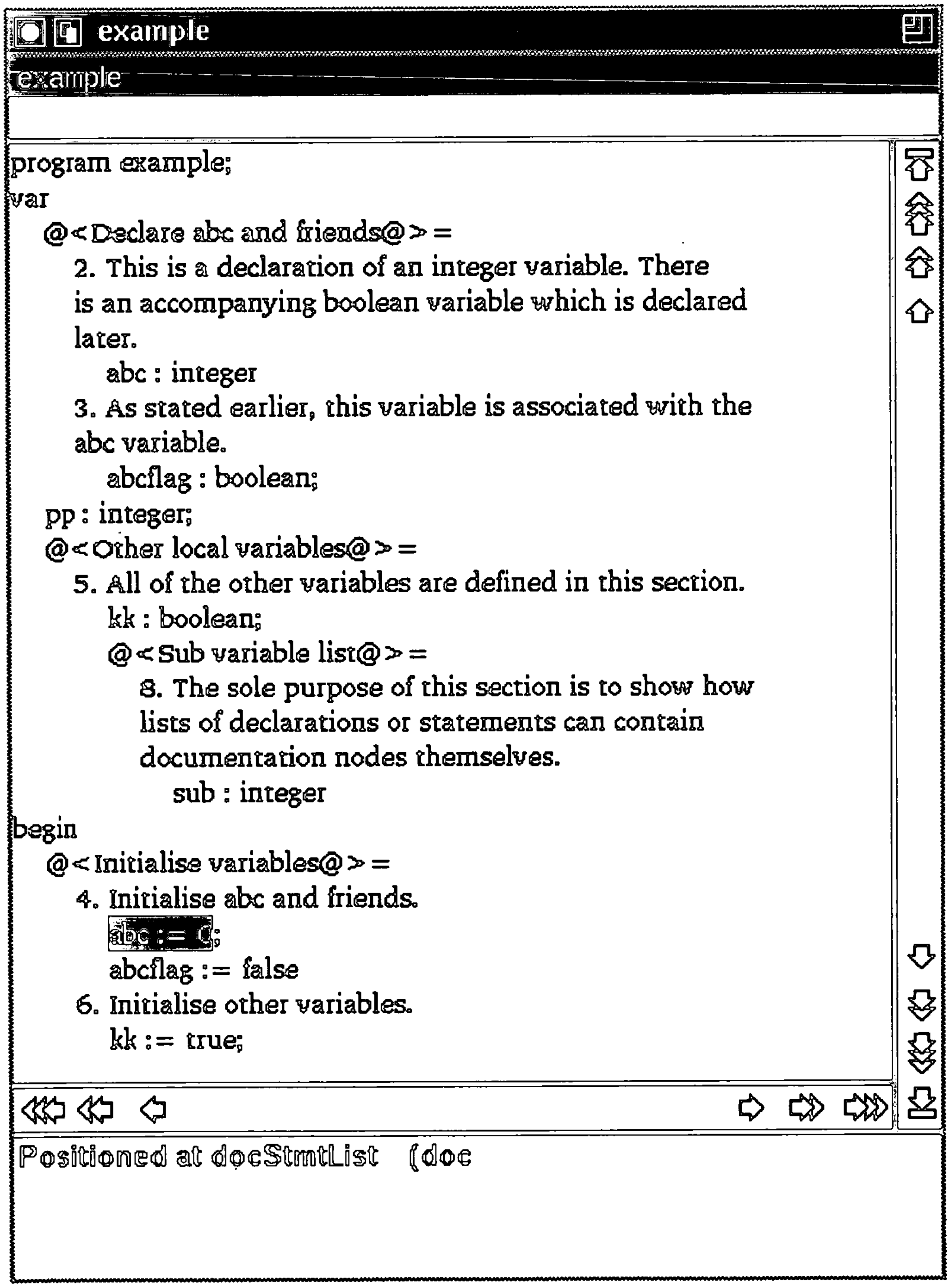
4.3 Feasibility studies

Two feasibility studies were performed to examine whether the Synthesizer Generator is capable of supporting these structures. The feasibility studies took an editor for the "toy" language presented in *The Synthesizer Generator* [97], and extended it to include documentation nodes using the two primary structures mentioned above. These feasibility studies were also used as an exercise in familiarisation with the Synthesizer Generator.

4.3.1 Program structure as primary structure

The first study used the program structure for the primary structure, and added documentation elaborations to certain points in the syntax. The documentation nodes in this editor were permitted at statement and declaration boundaries. The documentation nodes were displayed in a manner similar to WEB constructs. Each documentation node specified a section number, documentation string, and declaration or statement to enclose.

Figure 4–1 shows the display presented by the final version of this editor. The figure shows the general appearance of Synthesizer Generator based editors. At the top of the window (inside the window manager frame) is a reverse video bar, with the name of the buffer which is being edited; if an alternate view is displayed, the view name is also shown. The main text of the program appears in a large pane underneath the title bar, in the fonts specified by a style declaration. In this case, a single variable-width font (Times-Roman) has been used. To the right and below the main text are scroll icons, which allow the user to pan about the display. Clicking on the icons at the right hand side allows the user to move the display up or down a line at a time, half a page at a time, a page at a time, or directly to the top or bottom of the file. The icons below the main text allow the display to be moved by small, medium, or large motions left and right.



```
program example;
var
   @<Declare abc and friends@>=
      2. This is a declaration of an integer variable. There
      is an accompanying boolean variable which is declared
      later.
         abc : integer
      3. As stated earlier, this variable is associated with the
      abc variable.
         abcflag : boolean;
   pp : integer;
   @<Other local variables@>=
      5. All of the other variables are defined in this section.
         kk : boolean;
         @<Sub variable list@>=
            8. The sole purpose of this section is to show how
            lists of declarations or statements can contain
            documentation nodes themselves.
               sub : integer
begin
   @<Initialise variables@>=
      4. Initialise abc and friends.
         abc := 0;
         abcflag := false
      6. Initialise other variables.
         kk := true;
```

```
Positioned at docStmtList  (doc
```

Figure 4–1: Main view of "toy" editor with program structure priority.

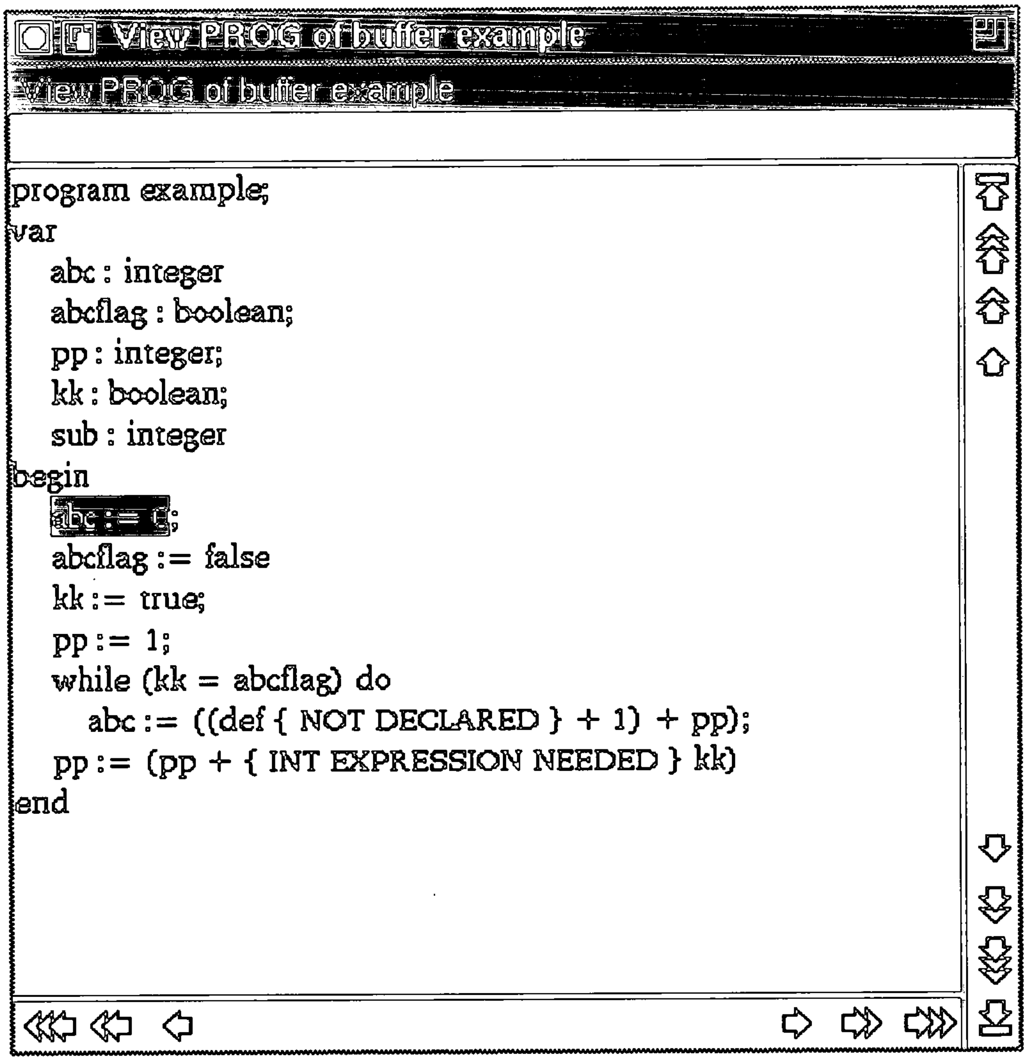
In editors created using the Synthesizer Generator, the part of the syntax tree being edited is called the *selection*. The selection can be moved using the mouse or cursor keys. Keys can be re-bound to provide different motion functions, but the normal orders of traversal available include widening the selection to the parent, moving forward or backward to sibling nodes, and pre-order traversal through the tree. Most of these motion functions are available in two varieties, which do and do not insert placeholders for optional nodes in the tree. Placeholders are markers which are inserted at points in the syntax tree where more complex structures can be placed. Optional placeholders are shown as the selection is moved to places where nodes could be inserted, and removed as the selection is moved on to another node.

The current selection in figure 4–1 example is an assignment statement; it is shown as white text on a black background. The pane at the bottom of the window shows the abstract syntax node at which the selection is positioned, and a series of transformation names. These transformation names may be selected to perform the transformation required on the current selection.

This editor went through several major versions as various methods of implementation in SSL were tried. The Synthesizer Generator documentation does not provide guidance with implementation methods or efficiency, and there are many areas in which the documentation does not specify the behaviour of the system clearly, so several trial implementations had to be discarded or modified significantly.

The final version of the editor was provided with three views of the program. These views corresponded to the annotated program, the raw program tree, and the documentation view. Views are selected by selecting the "Change View" option of the "Windows" editor menu, and typing the name of the view which is to be selected into the dialogue panel. A more direct method for selecting views is desirable.

Figure 4–2 shows the program view of the same program provided by this editor; the documentation information has been folded away, allowing direct access and editing of the program tree.



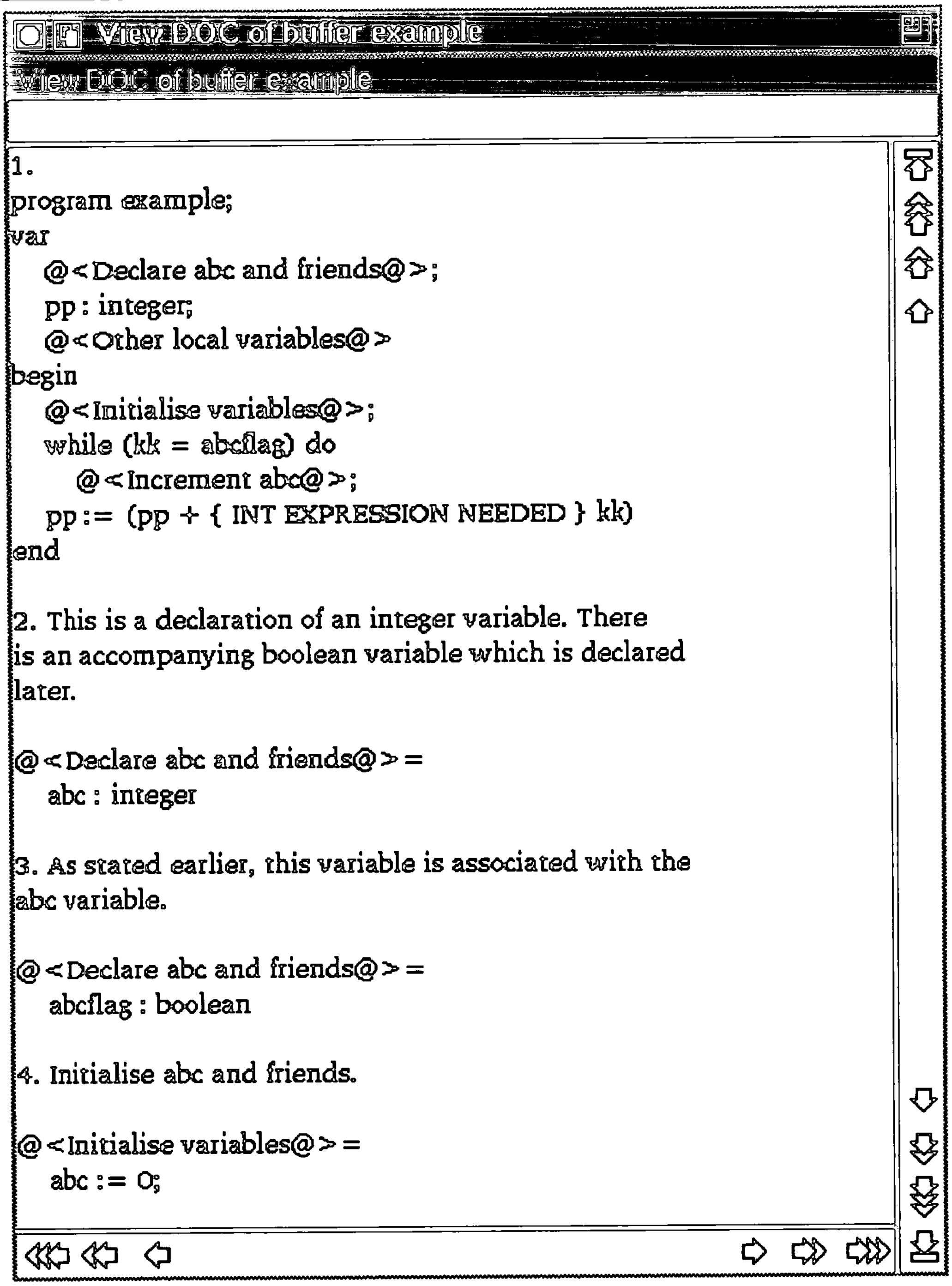
```
program example;
var
   abc : integer
   abcflag : boolean;
   pp : integer;
   kk : boolean;
   sub : integer
begin
   abc := 0;
   abcflag := false
   kk := true;
   pp := 1;
   while (kk = abcflag) do
      abc := ((def { NOT DECLARED } + 1) + pp);
   pp := (pp + { INT EXPRESSION NEEDED } kk)
end
```

Figure 4–2: Program view of "toy" editor with program structure priority.

The documentation view of the program is shown in figure 4–3. The sections of the main view have been re-arranged in the order of their section numbers (with the root section defaulting to number one), and are displayed in a manner similar to WEB.

This first feasibility study indicated where some of the major problems in using the Synthesizer Generator to create a literate program editor lie.

The facility to work with the documentation view of the program is essential for a literate program editor. This provides the essential feedback that indicates that the user is editing a *document* for human reading, and not just a program for machines to translate. Unfortunately, editing the documentation view of the



```
1.
program example;
var
   @<Declare abc and friends@>;
   pp : integer;
   @<Other local variables@>
begin
   @<Initialise variables@>;
   while (kk = abcflag) do
      @<Increment abc@>;
   pp := (pp + { INT EXPRESSION NEEDED } kk)
end

2. This is a declaration of an integer variable. There
is an accompanying boolean variable which is declared
later.

@<Declare abc and friends@> =
   abc : integer

3. As stated earlier, this variable is associated with the
abc variable.

@<Declare abc and friends@> =
   abcflag : boolean

4. Initialise abc and friends.

@<Initialise variables@> =
   abc := 0;
```

Figure 4–3: Documentation view of "toy" editor with program structure priority.

program in not possible with the program-structure priority editor, because the visible representation is generated from displayed attributes of the parse tree. The displayed attributes are generated by a function which sorts sections into the display order, from a list of sections created by a pre-order traversal of the parse tree. The SSL language operates in a functional manner, generating new copies of the phyla contained in each section, and in doing so, loses track of the correspondence between the displayed attribute and the original syntax tree. It is not possible even to select separate units within the documentation view of the program; either the whole buffer is selected or nothing.

This trial implementation has some of the flexibility that a complete tool would require; documented declaration and statement lists can be nested indefinitely. The price paid for this flexibility is a great increase in the complexity of the SSL description; *e.g.* declaration lists increase from one layer deep to four layers deep of structure. Pattern matches for transformations become extremely complex and lengthy, and threading attributes through all of the extra layers of structure becomes very cumbersome.

The editor shows a simple example of how the editor might be extended to check the static semantics of the program. Figure 4–2 shows two errors in variable usage which the editor has identified. The error messages appear in all views of the program, and will automatically disappear when the mistakes are corrected.

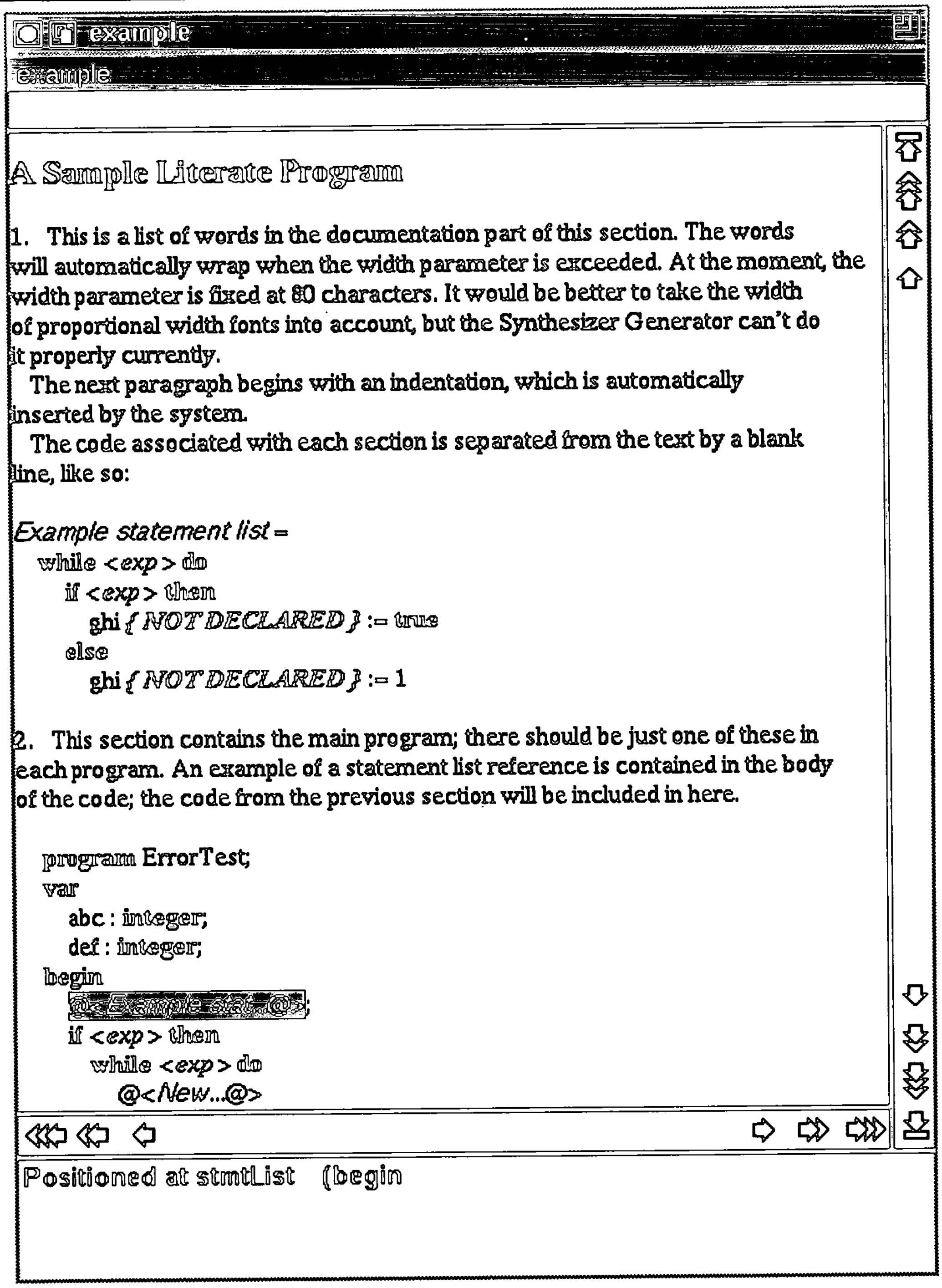
An editor using this idiom could not be described as "literate", because it lacks features essential to the paradigm *i.e.*, flexible order of presentation.

4.3.2 Documentation structure as primary structure

The second feasibility study used the documentation for the primary structure of the editor.

Figure 4–4 shows how this editor was displayed. More attention was paid to the appearance of this display than the previous example, using the Synthesizer Generator's display styles to differentiate parts of the program and documentation.

This implementation was extended to demonstrate some of the features that a full scale literate programming structure editor would require. The documentation parts were structured as lists of paragraphs, each of which was a list



A Sample Literate Program

1. This is a list of words in the documentation part of this section. The words will automatically wrap when the width parameter is exceeded. At the moment, the width parameter is fixed at 80 characters. It would be better to take the width of proportional width fonts into account, but the Synthesizer Generator can't do it properly currently.
The next paragraph begins with an indentation, which is automatically inserted by the system.
The code associated with each section is separated from the text by a blank line, like so:

```
Example statement list =
   while <exp> do
      if <exp> then
         ghi { NOT DECLARED } := true
      else
         ghi { NOT DECLARED } := 1
```

2. This section contains the main program; there should be just one of these in each program. An example of a statement list reference is contained in the body of the code; the code from the previous section will be included in here.

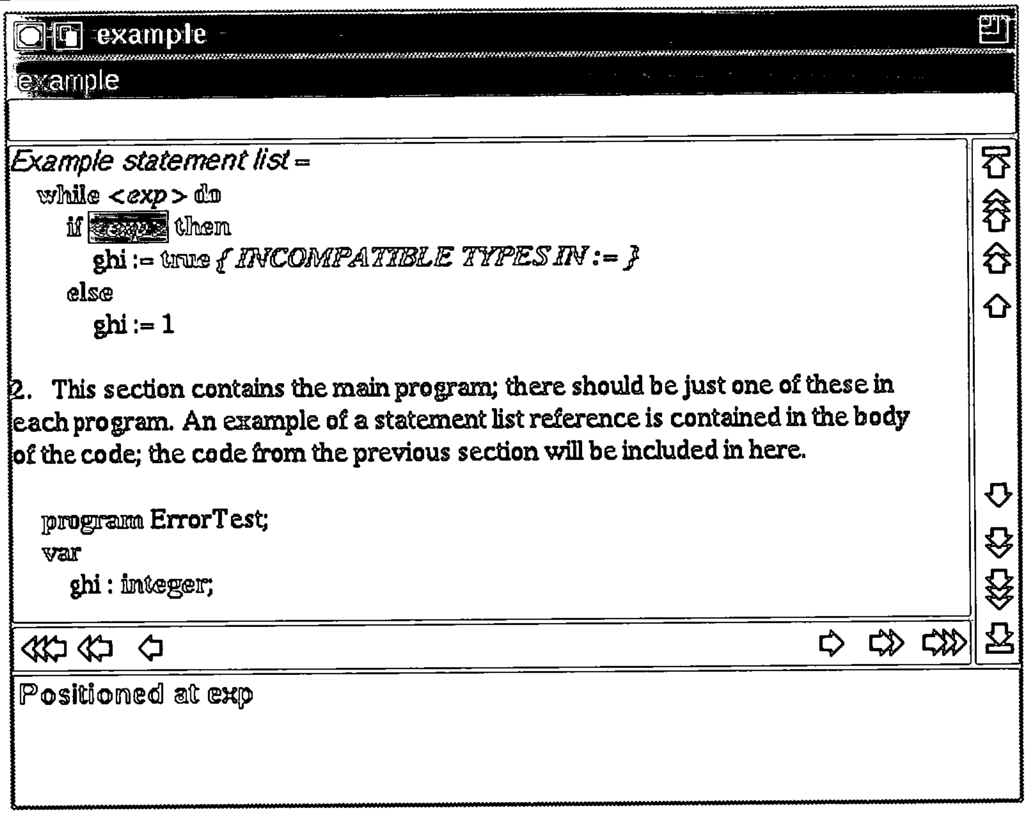
```
   program ErrorTest;
   var
      abc : integer;
      def : integer;
   begin
      @<Example stat...@>;
      if <exp> then
         while <exp> do
            @<New...@>
```

Positioned at stmtList (begin

Figure 4–4: Main view of the "toy" editor with documentation structure priority.

of words. This is a simple example of the abstraction of text formatting; having defined paragraphs as word lists, it would be simple to extend them to include emphasised or highlighted word lists as well.

Syntax and some static semantic checking were also been implemented. Error messages were displayed in a bold-italic variant of the font, beside the structure which they referred to. In figure 4–4, the error message "NOT DECLARED" appears after the references to the variable *ghi* in the first section. If a declaration of *ghi* as an integer is added to the program module in section 2, the error message is automatically updated to reflect the declaration, as shown in figure 4–5. The second assignment to *ghi* is now correct, but the first assignment now contains a type mismatch which is flagged.



```
Example statement list =
  while <exp> do
    if exp then
      ghi := true { INCOMPATIBLE TYPES IN := }
    else
      ghi := 1

2. This section contains the main program; there should be just one of these in
each program. An example of a statement list reference is contained in the body
of the code; the code from the previous section will be included in here.

  program ErrorTest;
  var
    ghi : integer;
```

Positioned at exp

Figure 4–5: A type mismatch error occurring after correction of the declaration error in the "toy" editor with documentation structure priority.

The semantic errors which are detected and reported in this way are:

Multiple declarations. A variable was declared more than once.

Variable not declared. A variable was used but not declared.

Incompatible types. The types of two expressions which should be the same (*e.g.*, both sides of an equality test) were different.

Boolean expression needed. An integer expression was used in a context in which a boolean expression was required.

Integer expression needed. A boolean expression was used in a context in which an integer expression was required.

The "toy" language for which this editor was written does not include the concept of variable scope, so the intricacies of dealing with multiple contexts in which variables may be valid was not investigated.

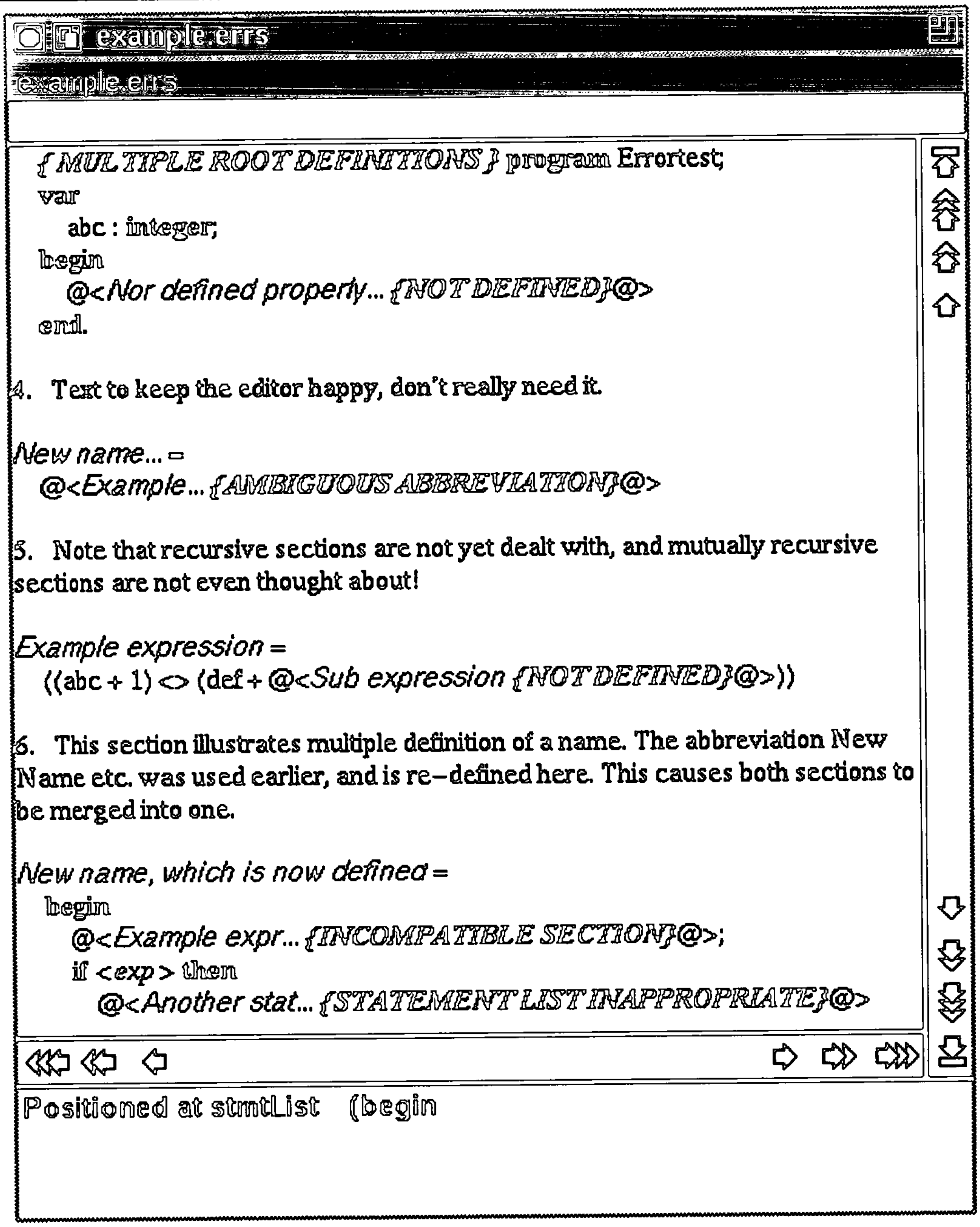
In addition to the static semantic checking, some syntax and documentation structure checking was implemented. The editor was designed so that the program contained a title (shown at the top of figure 4–4) followed by a list of sections. Each of these sections could contain a documentation part and an optional item which was either a root module (containing the top-level "program" declaration) or a named code fragment (an expression, a statement, or a statement list). The names applied to the code fragments could be sentences, including spaces, punctuation, and any other characters desired. Spaces in the section names were not treated as significant when comparing them, and they could be abbreviated by appending ". . . " to an unambiguous initial portion of the name. The sections were numbered automatically by the editor, and the section numbers were updated automatically when sections were inserted or deleted. It would be quite easy to extend the section numbering scheme to incorporate symbolic references to particular sections, via an aggregate which contained the labels and the section number.

Figure 4–6 shows some of the errors which can be detected in the syntax and documentation structure of the program.

The documentation and syntax errors detected were:

No title. The title of the program had not been defined.

Ambiguous abbreviation. An abbreviation which was not unique was used.


example.errs

example.errs

```
{MULTIPLE ROOT DEFINITIONS} program Errortest;
var
   abc : integer;
begin
   @<Nor defined properly... {NOT DEFINED}@>
end.
```

4. Text to keep the editor happy, don't really need it.

```
New name... =
   @<Example... {AMBIGUOUS ABBREVIATION}@>
```

5. Note that recursive sections are not yet dealt with, and mutually recursive sections are not even thought about!

```
Example expression =
   ((abc + 1) <> (def + @<Sub expression {NOT DEFINED}@>))
```

6. This section illustrates multiple definition of a name. The abbreviation New Name etc. was used earlier, and is re-defined here. This causes both sections to be merged into one.

```
New name, which is now defined =
   begin
      @<Example expr... {INCOMPATIBLE SECTION}@>;
      if <exp> then
         @<Another stat... {STATEMENT LIST INAPPROPRIATE}@>
```

Positioned at stmtList (begin

Figure 4–6: Documentation node errors in the "toy" editor with documentation structure priority.

No section name. A code fragment which required a name was created, but was not given a name.

Section not defined. A reference was made to a section which has not been defined.

Incompatible section. An expression-typed refinement was used in a context in which a statement or statement list refinement was required, or vice versa.

Inappropriate statement list. A statement list refinement was used in a context in which a statement was required. This case required that statements had an inherited attribute which indicated if statement lists were valid in the context.

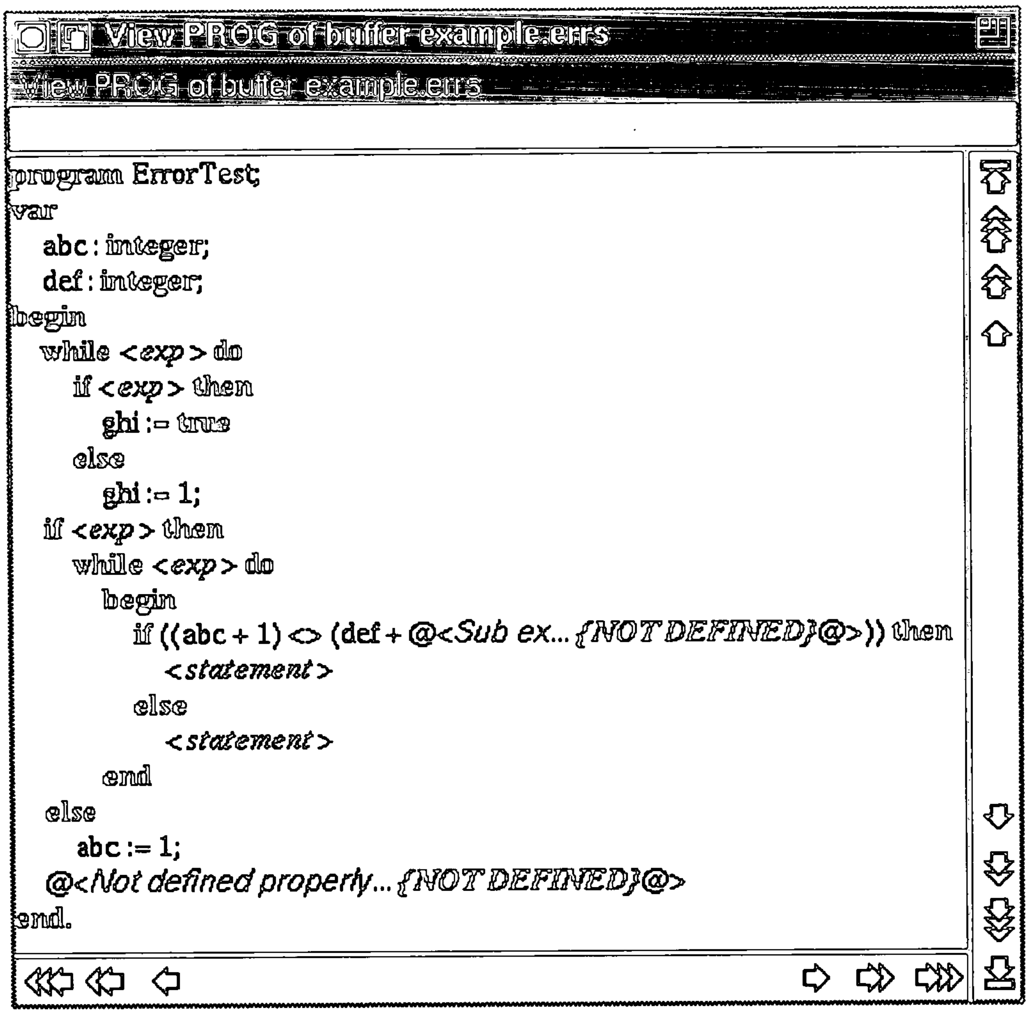
Multiple root definitions. More than one section contained a "program" declaration.

References to named code fragments could be placed at any point where an expression, statement, or statement list was required.

An alternative view of the documentation structure editor was provided, showing the program tree of the literate program. This view is shown in figure 4–7. Code references have been expanded, unless an error which prevented the expansion occurred. This view had to be implemented by a set of functions which used the Synthesizer Generator's pattern matching ability to deconstruct and reconstruct the program's parse tree. The pattern deconstruction and reconstruction operators had to be provided for every production from the "program" level down to the lowest level at which code references were allowed, in this case the expression level. This process is very tedious, and would not scale up to a large grammar very well.

The semantic errors detected and displayed in the main view of the program were not displayed in the program view, because the Synthesizer Generator did not fully attribute the program tree generated by the reconstruction functions.

The program view in this editor could only be used for browsing the program, and not for editing it; this was not a particular problem, because it encouraged the use of the literate view of the program, which was editable. There was no cross-referencing between the program view and the documentation view of the literate program. The facility to see which part of the documentation structure a piece of the program corresponds to would be useful for debugging


View PROG of buffer-example.errs
View PROG of buffer-example.errs

```
program ErrorTest;
var
   abc : integer;
   def : integer;
begin
   while <exp> do
      if <exp> then
         ghi := true
      else
         ghi := 1;
   if <exp> then
      while <exp> do
         begin
            if ((abc + 1) <> (def + @<Sub ex... {NOT DEFINED}@>)) then
               <statement>
            else
               <statement>
         end
   else
      abc := 1;
   @<Not defined properly... {NOT DEFINED}@>
end.
```

Figure 4–7: Program view of "toy" editor with documentation structure priority.

and walking through programs. The program view does allow the program to be written out in a compilable form, by selecting the "text" option when saving the file.

Table 4–1 shows the relative complexities of the original "toy" language editor, the literate program structure priority editor, and the literate documentation structure priority editor, as measured by the number of lines in each specification. The literate program editors are both significantly more complex than the original editor. The discrepancy between specifications for the attribute equations of the literate program editors is mainly due to the code which constructs the program tree shown by the program view. Minor differences in the documentation structure and completeness of the concrete syntax account for the differences between the unparsing, abstract syntax, and lexical analysis parts of the literate program specifications.

FUNCTIONS	FILE	ORIGINAL EDITOR	PROGRAM PRIORITY	DOCUMENTATION PRIORITY
Attribute equations	`p.m.ssl`	109	314	572
Unparsing	`p.u.ssl`	60	130	103
Abstract syntax	`p.x.ssl`	68	118	127
Lexical analysis and transformations	`p.y.ssl`	80	165	111
TOTAL		317	727	913

Table 4–1: Number of lines in each file of the “toy” language specifications for the original editor, literate program priority editor, and literate documentation priority editor.

4.4 Implementation of Synthesizer Generator editors

The trial implementations revealed several areas in which extra care is necessary in designing Synthesizer Generator specifications. The penalties for poor design in these areas are poor space and/or time efficiency, or extra complexity in the specification. The feasibility studies brought to light the major points to be considered when choosing between program and documentation structure as primary structure.

Some specific points are:

Use of list phylum SSL has a special type of grammar rule called a *list* phylum. These have certain advantages, such as built-in constructors to build and append to lists, automatic insertion of placeholders on some operations, and automatic transformation between singleton sub-lists and list elements. List phylum cannot cope with lists which have more than one element type, however; these have to be constructed out of lists of elements of one type, with these elements having alternative sub-types. Unfortunately, this approach loses some of the advantages of using a list phylum in the first place. Complicated lists can be constructed without using list phylum, but transformations have to be made available to perform the operations automatic to the list phylum.

List transformations If list phyla are used, transformations may be defined to operate on sub-lists or list elements (or even both). If transformations are defined to operate on sub-lists, they cannot be used on list elements, and vice-versa. Providing transformations to operate on both sub-lists and list elements is duplication of effort and increases the size and complexity of the generated editor.

Hierarchical/flat structuring Optional nodes of the editor's abstract syntax can be managed either by using SSL's `optional` declaration, or a new parent node can be inserted with and without the optional node. There are advantages and disadvantages to both methods, with the decision about which method to use being affected by the optional node's context; if the optional element affects the display representation of a node, or if the optional element appears in a non-optional context elsewhere in the specification, different strategies must be used.

Concrete input requirements The concrete input syntax provides the mechanism for translating between text and attributed structure. If pure text files are to be read by the editor, the entire concrete input syntax of the editor must be defined, with the associated problems of context sensitivity. Otherwise, decisions must be made about which parts of the abstract syntax should have concrete syntax provided. These decisions will greatly affect the usability of the generated editor, especially for experienced typists and programmers.

Synthesizer Generator inadequacies Some restrictions in the SSL language increase the complexity of the specification.

There are no global attribute variables in SSL, so references to remote attributes (even upward remote attributes of the root phylum) which are used in function declarations need to be passed in as parameters. If the function is used by other functions, each of the calling functions also needs to be passed the attributes as parameters.

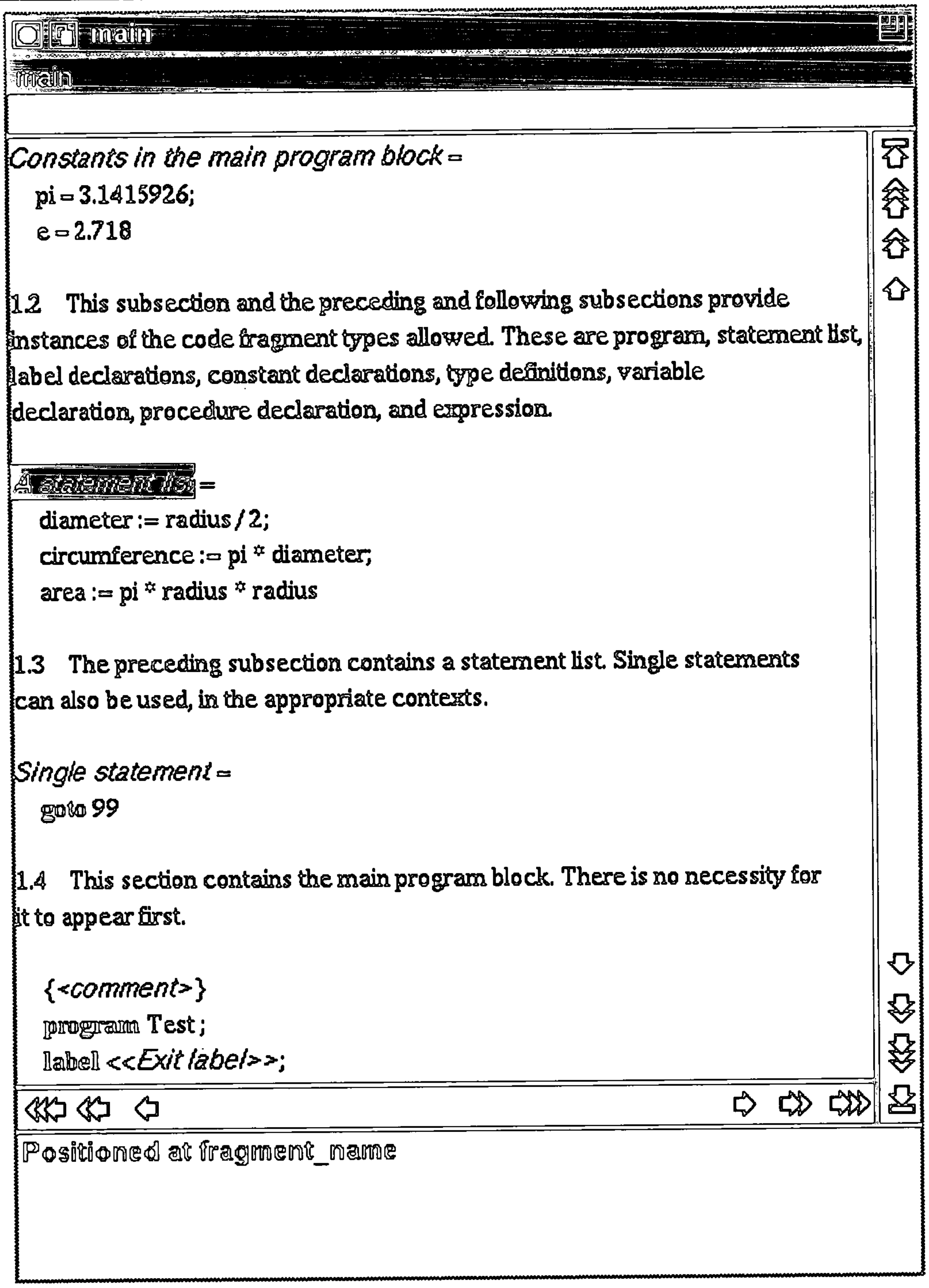
A particularly annoying restriction in SSL was encountered, which prevents the selection of attributes of phyla returned by function evaluations. This restriction significantly increases the complexity of editor specifications, because the work-around involves accumulating the attributes required, and threading the accumulated structure through the entire syntax tree to the nodes at which it is required. The error message output when this restriction is encountered ("`selection of an attribute of this sort of symbol not yet permitted`") indicates that the creators of the Synthesizer Generator are aware of the problem, but have not yet fixed it.

It was anticipated that more limitations of SSL would become evident when more complicated specifications were attempted.

4.5 The literate Pascal editor

The results of the second feasibility experiment were encouraging enough to warrant the development of a literate programming editor for a "real" language. Pascal was chosen because a fully-featured Pascal editor was provided with the Synthesizer Generator. Modifying this to incorporate literate programming was seen as an easier method than writing an editor from scratch. Most of the interesting problems from the literate programming side are not concerned with the exact details of syntax, so this course of action seemed very reasonable.

The Pascal editor was modified to include documentation nodes, with the documentation structure taking precedence over the program structure, as in the second feasibility study. The section structure was similar to the "toy" lan-



Constants in the main program block =
```
  pi = 3.1415926;
  e = 2.718
```

1.2 This subsection and the preceding and following subsections provide instances of the code fragment types allowed. These are program, statement list, label declarations, constant declarations, type definitions, variable declaration, procedure declaration, and expression.

A statement list =
```
  diameter := radius / 2;
  circumference := pi * diameter;
  area := pi * radius * radius
```

1.3 The preceding subsection contains a statement list. Single statements can also be used, in the appropriate contexts.

Single statement =
```
  goto 99
```

1.4 This section contains the main program block. There is no necessity for it to appear first.

```
  {<comment>}
  program Test;
  label <<Exit label>>;
```

Positioned at fragment_name

Figure 4–8: Main view of the literate Pascal editor

guage documentation priority editor, extended to include nested sub-sections as well as code fragments. Sub-sections could be nested to any depth. The documentation structure errors which were detected were similar to the "toy" editor: missing title and code fragment names, unmatched and ambiguous abbreviations, undefined code fragment names, and incompatible code fragments were checked.

Figure 4–8 shows a screen from the Pascal editor. The conventions of WEB were abandoned for the display format in this editor, because they introduced extra "noise" which was irrelevant with respect to using the editor.

The number of places at which code fragment references could appear was increased in the Pascal editor. References are allowed in label, constant, variable and procedure declarations, type definitions, statements, and expressions. References to statement lists were essentially the same as references to statements, except that an attribute was inherited which indicated that the context in which the reference appeared could support a statement list. The number of levels at which references can be made has an effect on how literate programs are expressed with the editor. If too few levels are provided, programmers will not be able to explain the program at the granularity that they want; on the other hand too many levels will complicate the editor unnecessarily without any gain in expressability, and may cause confusion about exactly what part of syntax a reference abstracts. An interesting point to note is that the representation of the code fragments may not necessarily be textually correct when rearranged into the correct order. As an example, consider the case in which a refinement is made for label declarations. In WEB parlance, the label declaration statement looks something like this:

```
label @<Labels in the outer block@>;
begin
  { ... }
```

We then have the refinement defined elsewhere:

```
@<Labels in...@>= 1, 99
```

If this refinement is later augmented by another definition:

```
@<Labels in the...@>+= 2, 3
```

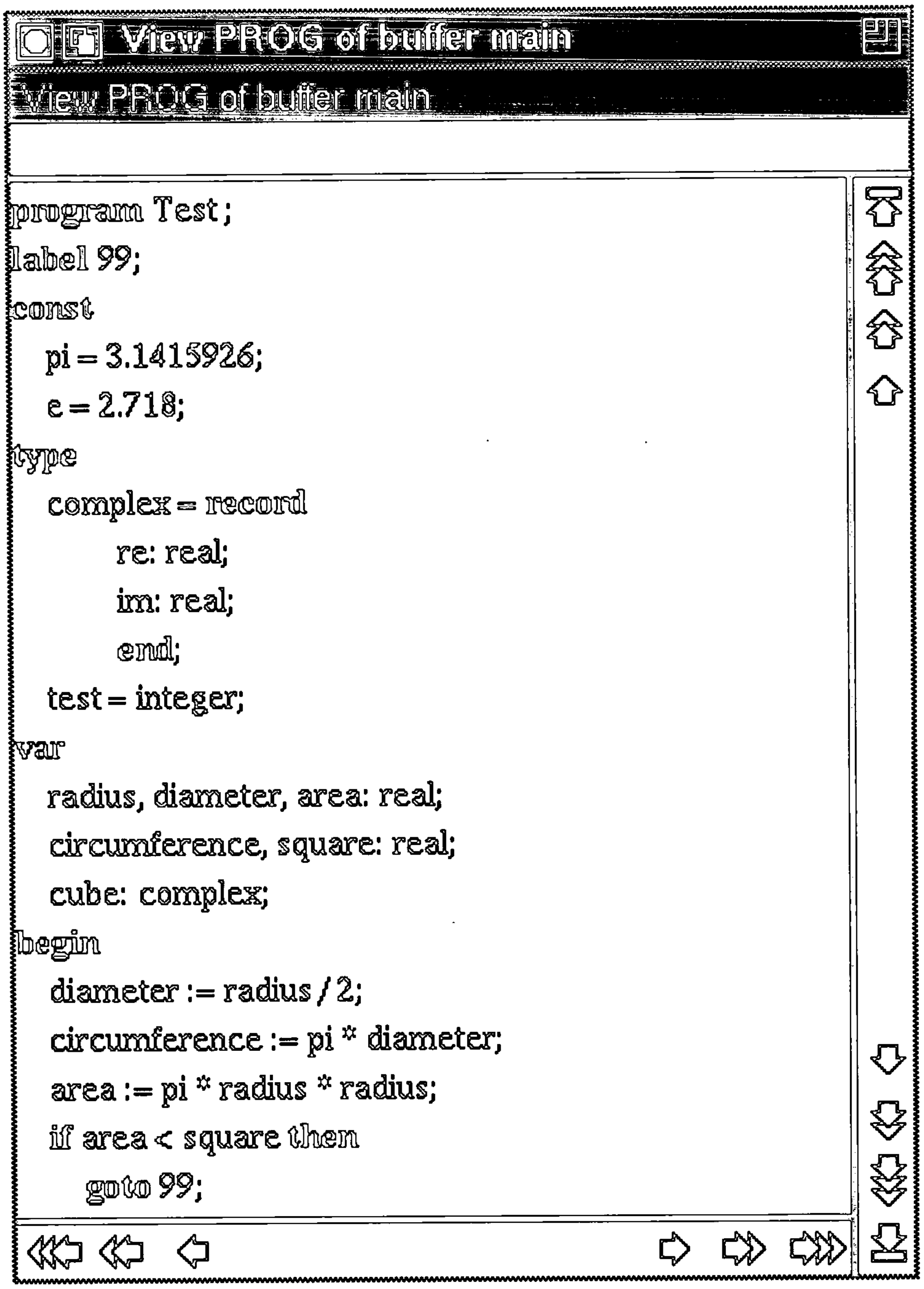
the resulting textual representation of the expanded refinement (`1, 99 2, 3`) is syntactically incorrect. To make the representation textually consistent, it would be necessary to allow either a dangling comma at the end of the refinement, or a preceding comma on the augmentation. Neither of these options is desirable, because they reduce the modularity of the code fragments: the refinement and augmentation cannot be viewed and edited (as lists of labels) separately. This point is not restricted to labels in Pascal; it will appear wherever lists in which the elements are *separated* by delimiters are used, rather than lists in which the elements are *terminated* by symbols.

In fact, this turns out not to be a problem for the structure editor, because the augmentation can be done by joining two lists of labels internally. The resulting internal data structure will have the correct representation when the display algorithm is applied to it for final output. It is desirable to keep the displayed representation consistent with the syntax of the language, but this is one case where doing so would impair comprehension of the program.

A program view of the Pascal editor was provided using the parse tree deconstruction and reconstruction technique developed for the second feasibility study. The program view is shown in figure 4–9. The difference in the the complexity of the Pascal grammar over the toy grammar shown by the increase in the work required to provide this view; 23 new functions were needed to perform the parse tree deconstruction and reconstruction.

The Pascal editor was already provided with more sophisticated semantic checking than the “toy” language editors. Full type checking was provided, as well as checks for duplicated declarations, parameters, record fields and case labels, and other miscellaneous errors. Adapting the semantic checking for the literate Pascal editor revealed some problems with the Synthesizer Generator.

The Pascal language incorporates the concept of static scope to restrict the visibility of names for variables, constants, types and procedures. Names can be reused in different contexts, with the most recently declared value for a name taking precedence over other values. Declarations can be made before the body of the program, or before the body of each procedure or function.



View PROG of buffer main

```
program Test;
label 99;
const
  pi = 3.1415926;
  e = 2.718;
type
  complex = record
        re: real;
        im: real;
        end;
  test = integer;
var
  radius, diameter, area: real;
  circumference, square: real;
  cube: complex;
begin
  diameter := radius / 2;
  circumference := pi * diameter;
  area := pi * radius * radius;
  if area < square then
     goto 99;
```

Figure 4–9: Program view of the literate Pascal editor

In the Pascal editor, the declaration structure was modelled using a MAP type, which associates values with names and is efficient for looking up the value associated with a name. The MAP was passed through the declaration lists (for constants, types, variables, and procedures) and augmented at each declaration. The inheritance structure of these MAPs was used to restrict the scope of names to the appropriate parts of the parse tree. The resulting MAPs were referenced from the body of the program or procedure by *upward remote attribute references*. This is a method of referring to the value of an attribute which is guaranteed to occur in a production which derives the production or phylum in which the reference occurs, without explicitly writing an inheritance declaration for each intermediate production. (Upward remote attribute references can actually refer to a list of productions or phyla in which the attribute occurs, in which case the value of the attribute in the first such ancestor production or phylum in the parse tree will be used.)

In the literate Pascal editor the declaration and reference scheme had to be changed to accommodate the code and declaration references. For each reference, what was desired was to find the section containing the corresponding declaration fragment, pass the inherited declaration MAP through to this fragment, and extract the augmented declaration MAP out of the fragment and pass it out as the declaration reference's output MAP. This was achieved by associating an input environment and an output environment with each section name; the input environment was the declaration MAP inherited by the reference to the section, and the output MAP was the augmented declaration MAP synthesised by the declarations in the section. Passing the inherited declaration MAP into the relevant fragment was achieved using an *attribution expression*, which forces the attribution of an unattributed term. In the version of the Synthesizer Generator used, attribution expressions were not fully implemented, and a work-around which used an extra production to simulate the full attribution expression had to be used. These input and output environments were collected into two MAP attributes which were referenced from the root of the grammar through upward remote attribute references.

The upward remote attribute references also had to be altered, because the program or procedure block productions were no longer guaranteed to be ancestors in the parse tree. The only guaranteed ancestor production was the section, which conveniently lent itself to using the same MAPs as were needed to thread the declarations through the section and reference structure.

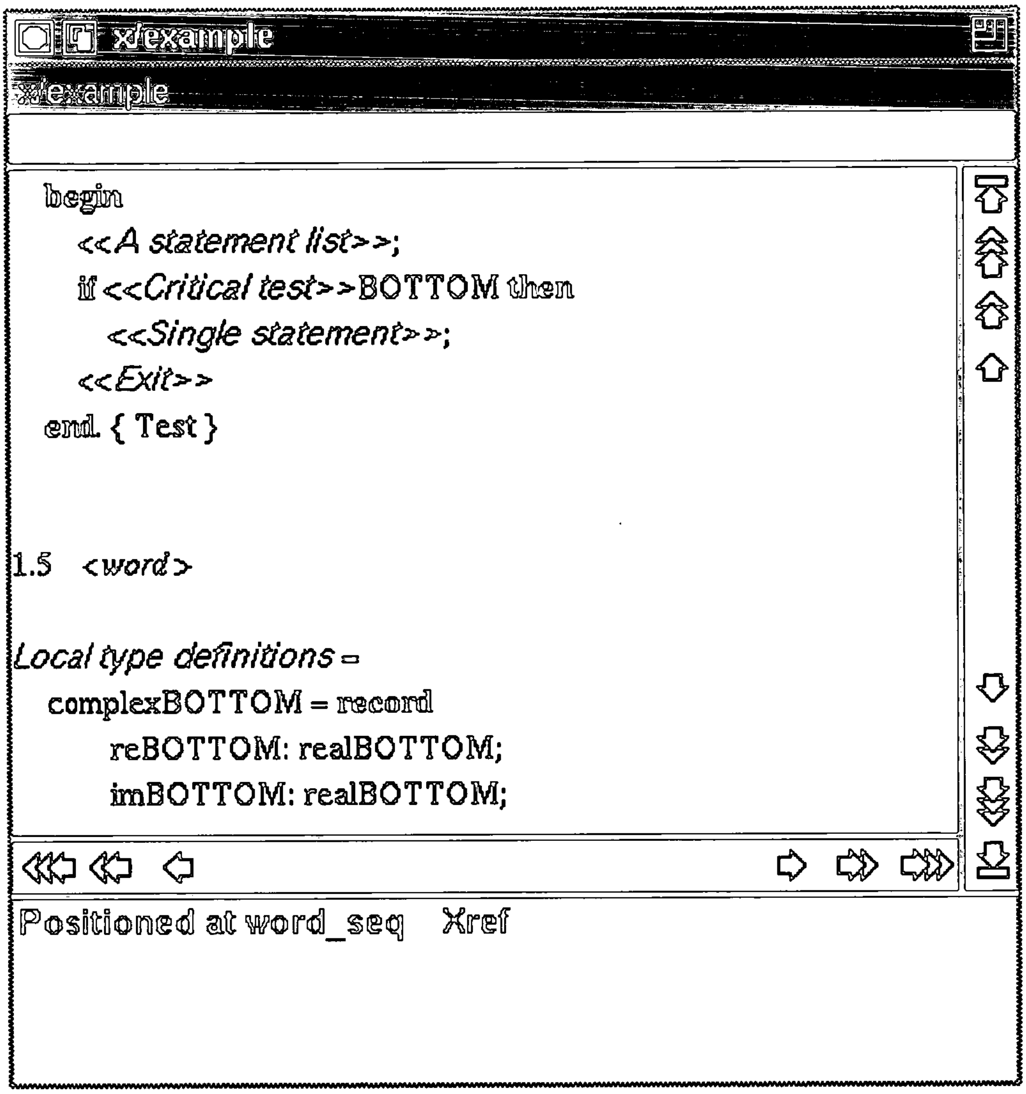
Unfortunately, these modifications introduce what is known as a type 2 circularity into the grammar. Grammars evaluated by the Synthesizer Generator's default method must be in the class of ordered grammars, which are non-circular. A type 2 circularity is a circularity in the approximation of the productions' transitive dependencies computed by stage 2 of Kasten's algorithm for orderedness [62], which is used by the Synthesizer Generator.

Kasten's algorithm sometimes reports type 2 circularity for non-circular grammars. In the case of the literate Pascal editor, the circularity is caused by the MAPs used to pass the input and output environments from sections to and from references. Neither of these MAPs may be computed in its entirety before the other, because each may depend upon values from the other MAP. The input environment to one section may be the output environment of another section, which in turn may take its input environment from the output of yet another section.

There is an experimental kernel in release 3 of the Synthesizer Generator which uses a method called *approximate topological ordering* to evaluate the attributes [55]. This method can be extended to evaluate grammars which are outside the class of ordered grammars. Compile time flags can be used to select the kernel and extend it to evaluate cyclic trees. Cyclic dependencies are evaluated by initialising the cycle with the completing term of an attribute instance on the cycle, and iterating around the cycle until a fixed point is reached. The implementation does not guarantee that the least fixed point of a cyclic dependency will be found.

The literate Pascal editor with the restructured declarations was compiled with the approximate topological ordering kernel and cyclic evaluation. It appeared to work when editing new programs, but problems were encountered with loading examples which had been saved previously. Figure 4–10 shows the display after loading a previously-saved example.

The displayed string "BOTTOM" is the external representation of the internal null-value attribute, which should never appear in a fully-attributed tree. It indicates that a part of the tree has been prepared for attribute evaluation,



```
  begin
    <<A statement list>>;
    if <<Critical test>>BOTTOM then
      <<Single statement>>;
    <<Exit>>
  end. { Test }


1.5  <word>

Local type definitions ≡
  complexBOTTOM = record
      reBOTTOM: realBOTTOM;
      imBOTTOM: realBOTTOM;
```

```
Positioned at word_seq   Xref
```

Figure 4–10: Attribution errors in the literate Pascal editor

but has not been visited by the evaluation algorithm, possibly because it has already been visited by a previous attribute evaluation.

In order for the literate program editor to be viable, this problem with circular dependencies must be resolved. A fully attributed program view would provide a work around the problem, because the semantic errors would be available through the program view. Unfortunately, not even this possibility is available with the Synthesizer Generator, because the program view provided by the parse tree reconstruction cannot be attributed.

4.5.1 Improvements to the literate Pascal editor

There are many improvements which could have been made to the literate Pascal editor, but were not done because of the fundamental problems with the Synthesizer Generator identified previously. Many of these improvements would be required before the editor could be contemplated as a useful programming tool:

- A complete concrete syntax, to allow importing complete Pascal programs and literate programs from other systems into the editor.
- Improvements to the documentation section structure. These would include emphasised, emboldened or highlighted words or phrases, automatic cross-referencing between sections, references for a global index, and maybe some generic structural markup templates for lists.
- Improved transformations for restructuring programs, which would make moving sections around the documentation hierarchy easy.
- Alternative views to provide input for text formatters to create a typeset version of the documented program.
- An index view, showing indices of identifiers and marked references, and a contents view, showing names of fragments or sections.
- An improved interface to the program views, using externally-linked functions to provide a pull-down menu of the views available.
- Methods for launching compilation or formatting processes from within the editor, after saving the appropriate view.

All of these improvements, with the possible exception of the last item, are within the capabilities of the Synthesizer Generator. The complexity of the SSL specification does have to be considered, though. The literate Pascal editor already requires recompilation of the Synthesizer Generator with larger values for several parameters, including the maximum number of phyla and the maximum number of productions allowed; the lexical analyser parameters also had to be increased. The increase in complexity of the specification for each of the improvements proposed may slow the editor's attribute recalculation down to the extent that it becomes unusable.

5

Evaluation

5.1 The literate program editors

A plan for implementing a literate program editor for a real programming language was outlined in chapter 3. Chapter 4 described the implementation of two literate program editors for a toy language, and one for the Pascal language. The implementation work corresponded to the implementation plan in the following way:

Stage 1. (Investigate how to represent literate programs with attribute grammars.) Two editors for a "toy" programming language were implemented, to test whether specifying the program or documentation structure as the primary structure was better. The latter method was selected as more appropriate, because the editable view of the program presented the documentation and code in the order in which the programmer intends the maintainer to read it.

Stage 2. (Create an editor for a real programming language.) A literate program editor for Pascal was implemented. Documentation and code could be inserted by selecting the appropriate point in the display and typing. No special commands were needed to switch between documentation and code or insert documentation, because the structure of the abstract syntax tree allowed easy movement to and insertion of sections with the normal motion commands. Sections were automatically renumbered when new sections were inserted, and the structure of the documentation and code was made obvious from the screen display. The list of transformation names at the bottom of the editor window can be used to give cues about using the editor; transformations to create documented sections or restructure code and documentation can have names which explain their purpose.

Stage 3. (Semantic checking of a real programming language.) Static semantic checking was implemented in the Pascal editor, but problems with the attribute grammar representation of the literate program prevented it from working correctly.

Stage 4. (Addition of publishing tools.) Publishing tools such as tables of contents, module cross references and index of identifiers were not implemented, although these particular examples would not have been particularly difficult to do, either as extra views of the program or automatically placed before and after the main program.

Even though the literate program editors created in the implementation studies were not complete, it is still instructive to compare their facilities to the characteristics of a literate programming system proposed by Detig and Schrod (see section 2.1.2).

Integration. In all of the editors the program code and documentation were contained within one file, fulfilling the requirement for verisimilitude. In the program-structure priority feasibility study, the presentation of the program tended to obscure the documentation to the extent that the immediate feedback between the formal statement of the algorithm (the program) and the informal statement of the algorithm (the documentation) was lost. This particular editor also allowed changes to be made within the program view, ignoring the documentation completely.

The provision of a program view in the literate program editors together with the facility to write this view out as a text file allowed direct compilation of the output of the editor.

Order of exposition. All of the editors created allowed the literate programs to be presented in a natural order of exposition. There were some restrictions on the program-structure priority toy language editor which could have been overcome by some redesign of the specification (*i.e.*, the main program fragment always appeared in the first section, and sections without program fragments were not permitted), but this editor also had the overriding flaw that the program could not be developed using the documentation view provided. The other editors allowed the development of the program in the most natural order for its exposition.

Refinements. All of the editors created allowed arbitrary names for the program fragments in refinements. The documentation structure priority editors allowed abbreviation of fragment names when using or adding to a refinement, but this was not necessary in the program structure priority editor, because the name was used only once, and the re-arrangement for documentation display propagated the name to all of the points where it was required.

In all of the editors, there were syntactic restrictions on where refinements could be positioned. These restrictions did not prove to be a problem, because enough places to expand refinements were provided. The restrictions ensured that the program remained syntactically correct within each program fragment, and removed the possibility for text macro expansion problems.

Publication tools. The editors created were noticeably lacking in this characteristic of literate programming systems. The reason for this is that other problems diverted the effort put into the editors. The features mentioned by Thimbleby (cross-references, indices and table of contents) would actually be the easiest to add to editors created by the Synthesizer Generator; the unparsing specification would need to be updated with displayed attributes providing this information, which can be collected through synthesised attributes of the documentation and program trees.

The editors created were used to examine the added benefits that can be gained from syntax-directed editing; the syntax of the program can be checked automatically, semantic errors can be detected, and inappropriate use of the refinement mechanisms can be caught easily, without requiring a separate compilation step.

The resolution of the problems identified with the literate Pascal editor in chapter 4 is fundamental to the success of the literate program editor. In order to be valuable enough to provide an incentive to use it, the literate Pascal editor has to provide more than just a text-based editing facility; the extra capabilities of a structure editor should be exploited to the full. The next section will examine improvements to the Synthesizer Generator which might have affected the outcome of the implementation studies. The rest of the chapter will examine the problem of circularity in the attribute grammar description of literate programs.

5.2 Relevant Synthesizer Generator improvements

The implementation work was performed using version 3.3 of the Synthesizer Generator. Since then, there have been three releases of the Synthesizer Generator, versions 3.4, 3.5 and 4.0, each of which have enhanced its capabilities.

Release 3.4 of the Synthesizer Generator permitted the use of alternative scanners and parser generators, instead of *lex* and *yacc*. *Yacc* has fixed table sizes, which cannot be altered without modifying the source code and recompiling, whereas *lex* reads a file giving alternative table sizes. Different analysers and parser generators which dynamically alter table sizes can be used (such as *flex* and *bison*). These improvements are important if the extensions to the literate Pascal editor suggested in chapter 4 are developed, simplifying the compilation of the editor specification.

Release 3.4 also included two unsupported packages which facilitate implementation of some of the improvements suggested for the literate Pascal editor. These were a package which permits the use of Athena widget control panels with the X11 interface to the editor, and an interprocess communication package to allow communication with other tools.

The work-around for the problem of displaying semantic information in the literate Pascal editor by providing an attributed program view can be implemented in release 3.5 of the Synthesizer Generator. This release provides *higher-order attribute grammars*, in which parts of the abstract syntax tree can be attributed. If these parts (known as nonterminal attributes) appear in the display of a buffer, the nonterminal attribute will be attributed and the unparsing specification can use its attributes. Nonterminal attributes are read-only, so a program view provided using this method will still not be editable, but it can be browsed using the normal editor movement commands.

Release 4.0 of the Synthesizer Generator is mainly a user-interface change, to make the interface compatible with the X Window system conventions. Better support is provided for creating menus and dialogue boxes to control the editor. Unfortunately, release 4.0 lacks the Approximate Topological Ordering attribute evaluation kernel which was used to compile the literate Pascal editor. This kernel was removed from the software to meet the release deadline, and may be included in future releases.

The later releases of the Synthesizer Generator still do not solve the problems encountered in the implementation of the literate Pascal editor. The nonterminal attributes introduced in release 3.5 make a work-around possible, at the expense of losing immediate feedback. The major problem with the literate Pascal implementation is that the attribute dependencies are cyclic. The secondary problem is that the implementation of cyclic attribute dependencies in the Approximate Topological Ordering kernel does not work.

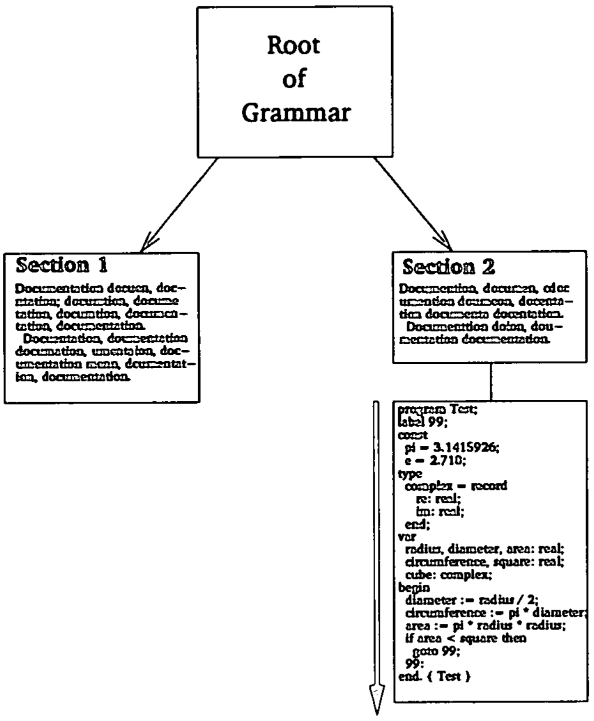
5.3 Cyclic attribute dependencies

The problem of cyclic attribute dependencies occurred in the implementation of the literate Pascal editor when semantic information was passed through the literate program in the natural order for the program, rather than the natural order for the documentation. In general, any attribute which depends upon the syntax tree of the program rather than the documentation will produce a cyclic dependency.

To see why this is so, we must consider how information is propagated around an attribute grammar. Attributes may either be inherited, which means that their values are derived from information held in their parent nodes, or synthesised, which means that their values are generated from information held in their children or the production in which the attribute appears itself. Information can only flow up and down the tree; attributes cannot depend on sibling production's attribute values, or on more remote relations' attribute values. Thus, in order to guarantee that an arbitrary node in the tree will have access to information generated from another arbitrary node, the information has to be passed up to their closest common ancestor by synthesising attributes, and back down again through inherited attributes. The closest common ancestor that can be guaranteed for two arbitrary nodes of the tree is the root of the grammar, since the nodes may be derived from separate parts of the root production. The problem then becomes one of mapping another tree structure (the program structure) onto the nodes of the grammar (the documentation structure), without producing cycles in the root node's attribute dependencies.

In the simplest case, one node of the documentation tree contains the entire program tree. In this case, all of the information which depends upon the program tree can be passed within the program tree itself, without reference to the documentation tree which encloses it. This situation is illustrated in figure 5–1. The thick arrow indicates the nodes of the documentation tree through which the information relating to the program has to pass. (The exact details of the program in this example do not matter; the example is showing the overall structure of the literate program.)

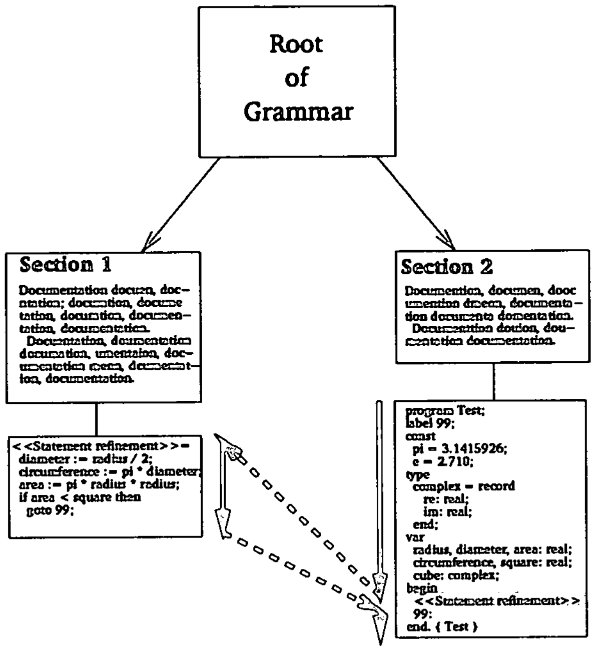
Each time a refinement is introduced into the program tree, a piece of the program node is removed, and inserted under a different node of the documentation tree. Information which depends upon the program structure needs



```
program Test;
label 99;
const
  pi = 3.1415926;
  e = 2.710;
type
  complex = record
    re: real;
    im: real;
  end;
var
  radius, diameter, area: real;
  circumference, square: real;
  cube: complex;
begin
  diameter := radius / 2;
  circumference := pi * diameter;
  area := pi * radius * radius;
  if area < square then
    goto 99;
  99:
end. { Test }
```

Figure 5–1: Information flow through program tree with entire program tree in one documentation node.

to be passed through attributes up to the root of the grammar from the point at which the refinement is used, back down to the node which contains the piece of the program tree, through the piece of program tree, up to the root of the grammar again, and back down to the point at which the refinement was used again, from where it can be passed through the rest of the program tree. The desired information flow through the program tree is illustrated in figure 5–2, and the actual information flow through the documentation nodes is illustrated in figure 5–3. The program fragment under the node labelled "Section 2" has a line with a grey background, indicating the *refinement reference* where the refinement which now appears under the node labelled "Section 1" was extracted from. The solid arrows indicate the flow of information through the code fragments, and the dashed arrows indicate the flow of information from the start of the refinement reference to the start of the refinement code, and from the end of the refinement code to the end of the refinement reference. Two attributes are introduced at the root of the grammar, one to pass the refinement its incoming information through to it, and the other to pass the refinement's outgoing information back to where it was used.


Root of Grammar
Section 1
Documentation docuzn, doc-ntation; docuation, docume tation, docuation, documen-tation, documentation.
Documentation, documentation documation, umentaion, doc-umentation meca, documentat-ion, documentation.
Section 2
Documention, documen, dooc umention dmeca, documento-tion documento domentation.
Documention doution, dou-mentation documentation.

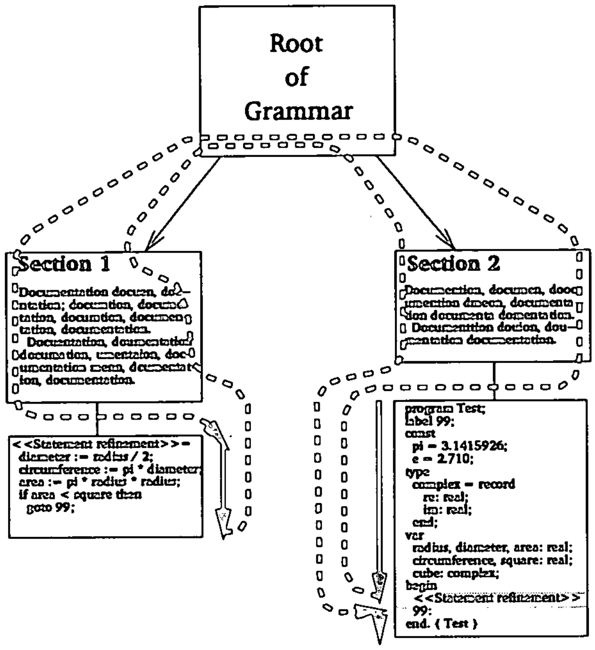
```
<<Statement refinement>>=
 diameter := radius / 2;
 circumference := pi * diameter;
 area := pi * radius * radius;
 if area < square then
   goto 99;
```

```
program Test;
label 99;
const
  pi = 3.1415926;
  e = 2.710;
type
  complex = record
    re: real;
    im: real;
  end;
var
  radius, diameter, area: real;
  circumference, square: real;
  cube: complex;
begin
  <<Statement refinement>>
  99:
end. { Test }
```

Figure 5–2: Ideal information flow through program tree with one documented program refinement.

This method of passing information is repeated for each refinement which is introduced. In order to avoid cyclic attribute dependencies we need to make sure that we have enough attributes at the root of the grammar to pass information through for all possible derivation trees. This can be trivially proven to be impossible. The number of attributes in the grammar must be finite, so the number of attributes at the root of the grammar must also be finite. Since there are no restrictions on the number of refinements which can be used, if we create one extra refinement when all of the attributes available for passing information to refinements are already used there will not be enough attributes to pass information for the new program tree.

This result implies that information which is passed through the program tree will necessarily need to share attributes at the root of the grammar. This can be done by using an array or list-valued attribute which accumulates information



```
<<Statement refinement>> =
  diameter := radius / 2;
  circumference := pi * diameter;
  area := pi * radius * radius;
  if area < square then
    goto 99;
```

```
program Test;
label 99;
const
  pi = 3.1415926;
  e = 2.710;
type
  complex = record
    re: real;
    im: real;
  end;
var
  radius, diameter, area: real;
  circumference, square: real;
  cube: complex;
begin
  <<Statement refinement>>
  99:
end. { Test }
```

Figure 5–3: Actual information flow through program tree with one documented program refinement.

from the various nodes which share it. Since these attributes are shared, there is no longer an explicit ordering of the portions of the attribute dependency graph which share these attributes. Some portions of the graph may require information from these attributes which other portions of the graph generate, and therefore there is a cyclic dependency on the shared attributes.

5.4 Convergent cyclic attribute dependencies

The fact that the attribute dependencies of a particular grammar may be cyclic does not necessarily mean that they cannot be evaluated; it merely means that they cannot be evaluated by a single pass over the attribute dependency graph. If we can guarantee that the values of the attributes on a cycle converge to a fixed point, we can evaluate the attributes. If we cannot guarantee convergence, then we run the risk of getting stuck in an endless loop if the attributes are evaluated.

In the case of a literate program editor, where a program tree is mapped onto the documentation structure grammar in an arbitrary order, the cyclic attribute dependencies are due to the arbitrary mapping of program tree nodes onto documentation tree nodes. Neither the program tree nor the documentation tree are themselves cyclic. The refinement process does not introduce cycles into the program tree, because refinements may not be used within themselves.

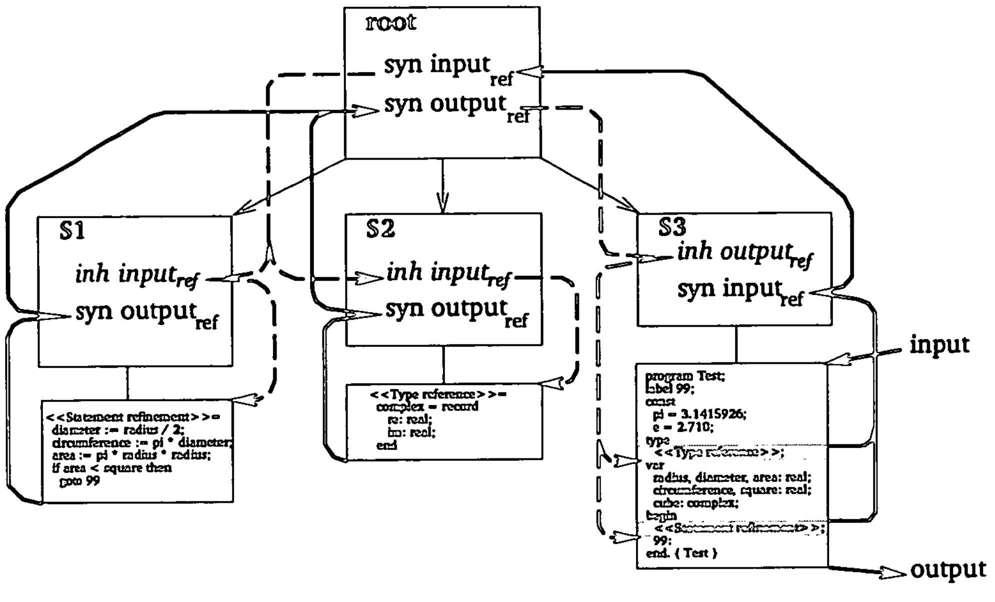
Both Farrow [41] and Jones [60] have shown how to construct fixed-point finding evaluators for circular attribute grammars. The conditions under which Jones's method work are slightly more restrictive than Farrow's: all circularly defined attributes must have semantic functions that are monotone and yield values from a domain forming a lattice of finite height. Jones's method involves finding the strongly connected components (SCCs) of the attribute dependency graph, and evaluating the fixed points of each strongly connected component in an order defined by a topological sort of the SCC graph. This ensures that each fixed point computation is performed only once. Jones presents a naive algorithm for evaluating the fixed points of each SCC which performs $O(k \sum_{i=1}^{k} h_{x_i})$ attribute evaluations in the worst case, where k is the number of attributes in the SCC and h_{x_i} is the height of the attribute instance x_i in the domain satisfying the fixed point equation (see [60]). Jones also presents a more efficient algorithm for evaluating the SCCs which performs $O(\sum_{i=1}^{k} h_{x_i})$ attribute evaluations in the worst case.

Farrow's method requires that the semantic functions on the attributes are monotone, yield values from domains forming complete partial orderings, and satisfy the *ascending chain condition*. Jones suggests that this difference in conditions is slight, because any partially ordered set can be embedded in a complete lattice.

In the case of the literate program editor, the attributes in the cycle are a set of shared MAPs which associate names of refinements with data passed into or out of these refinements, and a set of attributes within each refinement which contain information such as symbol tables, labels in scope, or types of expressions. Semantic functions on the MAP attributes are of two types; they either copy the value of the attribute through to the next node in the cycle (these are known as *copy rules* in attribute grammar literature), or they add a new name and associated information to a MAP. These semantic functions are monotonic, since a lattice can be constructed with its height bounded by the number of elements in a MAP (the domain of the function), and the functions never return a MAP with less elements than their argument MAP. However, since the height of the lattice is bounded by the number of elements in the MAP, and the MAP contains a finite, but unbounded number of elements, the lattice representing the domain of functions operating on MAPs is not of finite height. Thus Jones's method (and by implication, Farrow's method) for constructing fixed-point attribute evaluators will not necessarily work for the literate program editor. Failing to meet the conditions for Jones's algorithm does not mean that it will *not* work for the literate program editor; it does mean that the algorithm is not *guaranteed* to find a fixed point.

The attribute dependency graph in the literate program editor does have some special structural properties which can be used to prove that a fixed point exists, and to give a bound on the number of attribute evaluations which will be performed. Figure 5–4 illustrates how attributes are used to pass information through the program tree. The solid arrows indicate the flow of data up the tree via synthesised attributes; these are the input data to refinements, or the output data from refinements. The dashed arrows indicate the flow of data down to the leaves via inherited attributes; in the actual implementation, some of the copy rules are implicitly inserted by upward remote attribute references. The grey areas in the node labelled "S3" indicate where refinements were introduced.

The corresponding attribute dependency cycle is shown in figure 5–5; this is a strongly connected component of the whole attribute dependency graph. The dashed box encloses the shared MAP attributes. The dashed arrows show where the detail of the internal attribute structure of program nodes have been omitted for clarity; there will not be any cycles within the program node attributes, as they are constructed in the usual way. The dependency graph for the literate program editor is constructed in such a way that the attributes which pass data through the program will have a set of shared MAP attributes, and several cycles



```
<<Statement refinement>>=
  diameter := radius / 2;
  circumference := pi * diameter;
  area := pi * radius * radius;
  if area < square then
    goto 99
```

```
<<Type reference>>=
complex = record
  re: real;
  im: real;
end
```

```
program Test;
label 99;
const
  pi = 3.1415926;
  e = 2.710;
type
  <<Type reference>>;
var
  radius, diameter, area: real;
  circumference, square: real;
  cube: complex;
begin
  <<Statement refinement>>;
  99:
end. { Test }
```

Figure 5–4: Documentation tree of a literate program, showing inherited and synthesised attribute chains. *Italicised* attributes are implicit remote attribute references.

of attributes that inherit values from the shared MAPs and synthesise new values which are inserted into the shared MAPs. The set of shared MAPs will contain two MAPs at the root of the tree, one for the information inherited by the refinements and one for the information synthesised by the refinements. There is one attribute cycle for each refinement in the program. Figure 5–5 shows two cycles around the dependency graph, for the S1 fragment and the S2 fragment.

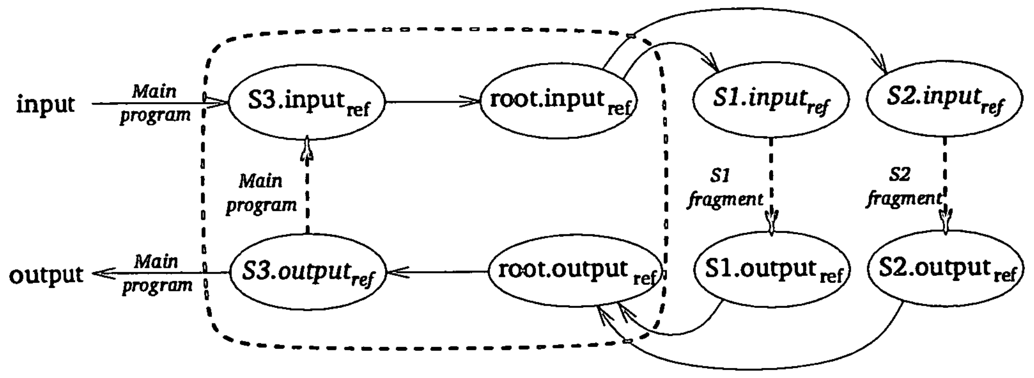


Figure 5–5: Attribute dependency graph of the program tree in figure 5–4, showing alternative cycles around the graph.

Note that the $input_{ref}$ and $output_{ref}$ attributes are aggregates which are indexed by the refinement names used in the different cycles around the dependency graph; thus evaluating a particular attribute cycle does not destroy the information generated by evaluating other cycles. The only inputs to the dependency cycle will be from the main program node, and the only outputs from the cycle will be to the main program node; this information can be used to decide where to start the attribute evaluation.

The properties of the attribute dependency graph which ensure that an evaluation will reach a fixed point are:

1. Each cycle can only depend on the values generated by cycles whose corresponding refinements appear before the cycle's corresponding refinement in a traversal of the program tree. This property follows from the way in which the dependency graph was constructed; a dependency between cycles indicates that an attribute which is passed through the program tree was used at two points in the program, separated by the refinements associated with the cycles.

2. No cycle on the graph alters the values generated by another cycle (this follows from the restriction of refinements to have different names).

3. No cycle depends on the value generated by itself (this follows from the restriction on refinements that they cannot appear in their own expansion).

Properties 1 and 3 indicate that the dependencies between cycles can be ordered. Properties 1 and 2 indicate that the value generated by a cycle will not change once the values on which it depends have been completely evaluated. Since the number of refinements in the program must be finite (or else the program would be infinite), there are a finite number of cycles corresponding to the refinements, and so a finite number of evaluations of these cycles will reach a point where all of the cycles have been completely evaluated.

Figure 5–6 shows an algorithm that will evaluate the fixed point of the dependency cycle. Since the cycles in the dependency graph SCC correspond to program refinements and the input to the SCC is from the main program node, the input attribute will be in the shared MAP attributes, and thus is an obvious candidate for the initial attribute. The root node of the literate program derivation tree will also contain shared attributes, so these attributes could also be considered for the initial attribute. All of the cycles in the dependency graph will pass through this shared initial attribute. The algorithm finds all of the cyclic paths through the SCC, and evaluates them in turn until the value of the initial attribute is consistent. Since the initial attribute is shared by all of the cyclic paths through the SCC, and contains information generated by all of them, it will only be consistent when all of the attributes in the cyclic paths (and hence the SCC) are consistent.

```
{ The set $a_g$ is the set of all attribute instances in the SCC $g$ }
initialise each attribute instance $x \in a_g$ to $\bot_x$ ;
select an initial attribute $i$ from the shared MAP attributes;
initialise set of cyclic paths $p_c$ through $g$ to $\varnothing$;

FIND_PATHS($i$, nil) { i.e., put all cyclic paths from $i$ through $g$ into $p_c$ };
repeat
  $i_0 \leftarrow i$;
  $p \leftarrow p_c$;
  while $p$ is not empty do
    select an remove a path $q$ from $p$;
    PATH_EVALUATE($q$);
  od
until $i = i_0$;
```

Figure 5–6: Algorithm for evaluating the fixed point of a strongly connected component of the dependency graph of the literate program editor.

The fixed-point computation will terminate because of the properties of the graph shown above. Each time all of the cyclic paths through the graph are evaluated (this will be referred to as a *full evaluation*), at least one cycle will have its dependencies satisfied, and will generate a correct value. The first refinement in the program does not depend on any other; after the first full evaluation the value generated by the cycle corresponding to it will be correct. The cycle corresponding to the second refinement in the program depends upon the value generated by the first refinement; after the second full evaluation it will also be correct. Since there are a finite number of refinements in the program, and the values generated by the cycles do not change once their dependencies are satisfied, there will be a finite number of full evaluations before the attributes reach a steady state, *i.e.*, a fixed point.

The evaluation of the attributes on a path through the dependency graph is shown in figure 5–7.

```
PATH_EVALUATE($p$) { $p$ is a path }:
  while $p$ is not nil do
    split $p$ into head $h$ and tail $t$;
    evaluate $h$;
    $p \leftarrow t$;
  od
end
```

Figure 5–7: Evaluating a path through an SCC of the dependency graph.

An algorithm for finding the cyclic paths through the dependency graph is shown in figure 5–8. The way in which the dependency graph is built guarantees that the every cycle on the graph will pass through the initial shared attribute i. This algorithm will terminate because every node in the SCC is part of a cyclic path (from the definition of an SCC), and following the successors of the node will lead back to the initial attribute.

The fixed-point evaluating algorithm evaluates each cyclic path from the initial attribute to its predecessor, on which the initial attribute depends. It is

```
FIND_PATHS($x$, $p$) { $x$ is an attribute, $p$ is a partial path }:
  $p \leftarrow p.x$ { append $x$ to $p$ };
  for each successor $y$ of $x$ in the SCC $g$ do
    if $y$ is the initial attribute $i$ then
      insert $p$ into $p_c$ { $p$ is a cyclic path };
    else
      FIND_PATHS($y$, $p$);
    fi
  od
end
```

Figure 5–8: Finding the cyclic paths through an SCC of the dependency graph

quite possible that in one full evaluation more than one cycle can reach their final values; if a cyclic path *b* depends on the value generated by another cyclic path *a* which has had its dependencies satisfied by the previous full evaluation, and *a* is chosen for evaluation before *b*, the values of the shared attributes will satisfy the dependencies of *b* when it evaluated. In the best case, all of the cyclic paths will be chosen for evaluation in exactly the order in which they depend upon each other, and only one full evaluation will be needed. In the worst case, the cyclic paths will be chosen in reverse dependency order, and only one cycle will generate its final value in each full evaluation. One full evaluation will be needed for each cycle in the dependency graph.

Since the number of cycles in the dependency graph corresponds to the number of refinements in the literate program, the best case running time of the algorithm is $O(n)$ attribute evaluations, where n is the number of refinements. The worst case running time is $O(n^2)$ attribute evaluations.

This compares quite favourably to the complexities of Jones's algorithms when applied to the literate program dependency graphs. The conditions that guarantee termination of the algorithms are similar. All of the cycles of the dependency graph will be evaluated on each pass of the algorithms, so the value generated by at least one cycle will reach a stable value after each pass.

Jones's naive algorithm only uses values generated by the previous pass over the attributes, so the value generated by at most one cycle will reach its final value in each iteration. The naive algorithm will always take the same time as the worst-case bound for the algorithm presented above, *i.e.*, $O(n^2)$ attribute evaluations.

Jones's efficient algorithm does not fare any better: each iteration performs a depth-first search of the dependency graph, marking nodes so that it can detect and prevent looping. On each pass, the value generated by at least one cycle will reach a stable point, but the value generated by only one of the cycles will be propagated through to the shared MAP attributes because of the node marking. The attributes which have been altered on the other cycles will be put into the set of attributes from which the evaluation may be started in the next pass. Since all nodes of the dependency cycle are reachable from any other, it does not really matter where the algorithm restarts from. The pruning test in this algorithm which prevents further evaluation does not help much, because of the MAP-valued attributes. If the value generated by any cycle is altered, it is put into a MAP with its corresponding refinement name. All of the MAP attributes on the cycle (*i.e.*, all of the shared attributes and some others) will be deemed to have changed, even though the value generated by a particular cycle may not have changed. The node marking which was intended to make this algorithm more efficient actually hinders it for the literate program graphs, because at the value generated by only one cycle will be propagated in each pass. The efficient algorithm will thus take the same time as the naive algorithm, $O(n^2)$ attribute evaluations.

There in an optimisation which can be applied to the algorithm presented above. Once the value generated by each cyclic path has reached its stable value, the path can be removed from the set of cyclic paths. From the termination conditions, at least one cyclic path will be removed from the set of cyclic paths in each full evaluation, improving the worst case running time to $O(\sum_{i=1}^{n} i)$ attribute evaluations. Unfortunately it is difficult to test if the value generated by a particular cycle has changed, for the same reasons the pruning test in Jones's efficient algorithm does not work: the internal structure of some attributes must be exposed to do so.

Implementing this algorithm in a literate programming editor poses some problems. It is impossible (because of uncomputability) to check the constraints on the semantic functions that guarantee termination. It has also been shown that testing for noncircularity of an attribute grammar is intractable [59]. Most attribute grammars that arise in practice are members of restricted classes of attribute grammars for which noncircularity is guaranteed, *e.g.*, ordered attribute grammars [62]. In order to check that the attribute grammar can be evaluated by the literate program editor, it may be necessary for the programmer to provide hints about how to treat various attributes. The shared MAP attributes could be marked by the implementor, and removed from the circularity test. Any other circularities detected could then be announced as errors.

The algorithm presented above operates on the strongly connected components of the attribute dependency graph. Algorithms for finding the strongly connected components (SCCs) of a graph are well known [108]. The SCCs of an attribute dependency graph can be altered radically by a single modification to the abstract syntax tree. Jones showed how SCCs could be identified on demand, with an overhead linear in the cost of computing the fixed points of the SCCs. The structure of the dependency graphs in the literate program editor are such that cycles will only occur when shared MAP attributes are used to pass information through the program tree. If these attribute were marked by the system implementor, as was suggested before, and cycles are not permitted in the rest of the dependency graph, the SCCs can be computed statically. A static evaluation algorithm can be used for evaluating the attributes which are not part of the SCCs, and the algorithm presented can be used to find fixed points of the SCCs.

There is a condition on the use of the algorithm presented in figure 5–6 that has not been mentioned yet, because it is satisfied by the dependency graphs generated by the literate program editor. This is that each strongly connected component should contain only one set of shared attributes which directly depend upon each other. Restated, all of the cyclic paths in the SCC must pass through at least one shared attribute. This condition could be broken if two separate attributes were passed through the program tree using their own shared attributes, and each attribute was dependent upon a previous value of the other attribute. Figure 5–9 shows a simplified view of an SCC of the dependency graph that would arise in this case. In the literate program editor, the condition is satisfied by constructing an aggregate phylum containing all of the attribute values which are passed through the program structure, and using a single shared MAP to pass this aggregate through the program. At the start of each refinement the values are separated out into their own attributes, and at the end of the refinement the attribute values are collected back into the aggregate. Figure 5–10 shows a simplified view of the dependency graph SCC generated by this method. If a single aggregate is passed through the program tree, there may be multiple paths through each cycle of the SCC, so the algorithm has to evaluate all of these paths in each full evaluation. This gives a running time of $O((n + x)^2)$ attribute evaluations for the algorithm, where n is the number of refinements in the program and x is the number of attributes collected into the aggregate. The corresponding complexity for x attributes evaluated in independent SCCs is $O(xn^2)$ attribute evaluations.

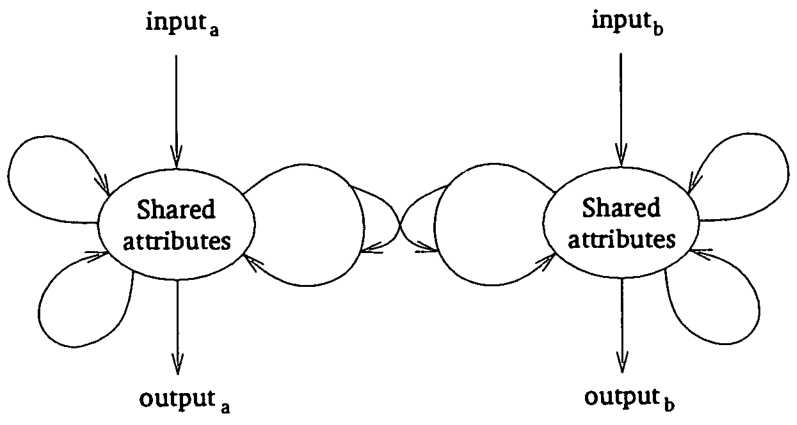


Figure 5–9: Simplified SCC of a dependency graph with mutual dependencies between two attribute chains failing conditions for algorithm 5–6

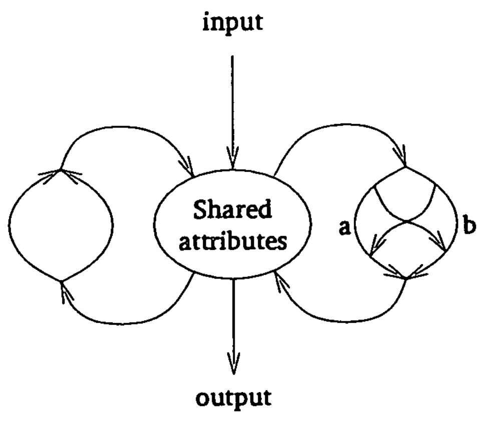


Figure 5–10: Simplified SCC of a dependency graph of aggregated attributes with mutual dependencies within refinements passing conditions for algorithm 5–6

The observation that the attribute dependency graphs created by the literate program editor can contain multiple cycles is a clue to the behaviour of the Synthesizer Generator when presented with the literate Pascal editor. The Synthesizer Generator uses an incremental attribute evaluation algorithm which is invoked whenever a change to the edited tree is committed. The algorithm uses a change propagation mechanism to limit the attributes which are considered for re-evaluation to those which are affected by the change. The change propagation mechanism sets the attributes which may be affected by the change to the internal null value, and then evaluates the attributes affected by the change. The cyclic tree evaluator provided by the approximate topological ordering kernel picks an attribute instance on a cycle, sets its value to the completing term for the attribute, and iterates around the cycle until a fixed point is reached. The appearance of the “BOTTOM” representation of the internal null value in the display is probably because the fixed point evaluation algorithm is either not handling multiple cycles correctly, or is pruning the attribute dependency graph incorrectly. Only part of the attribute dependency graph is evaluated, but the whole graph is initialised.

The algorithm provided here is quite specific to the attribute dependency graphs generated by the literate program editor; it is not likely to be applicable to graphs which are not generated in a similar way. The technique of finding evaluating cyclic paths through the dependency graph may be applicable to other cyclic attribute grammar evaluation methods, especially where attributes can take list or array values.

Proving that the literate program's attribute dependencies are convergent within a given number of attribute evaluations shows that a literate program editor could be implemented using attribute grammars, but there are performance implications which also have to be considered. In an interactive system, the delay introduced by the worst-case $O(xn^2)$ or $O((n + x)^2)$ evaluation of the fixed points of the attribute dependencies are likely to yield unacceptable response times. Two of the programs surveyed illustrate the potentially large number of attribute evaluations; an average sized program such as the Cweave processor has 210 sections, most of which contain refinements, and at the top end of the range a version of the TEX program contained 1377 sections.

What is required is a better model for representing literate programs, which can be used to build editors with all of the power available in attribute-grammar based systems, but without the cost of cyclic attribute evaluation.

6

Representing literate programs

6.1 Views of literate programs

Literate programs are complex structures; they can be used to present a program to a human reader in a comprehensible way, and to derive a program for a compiler to read. These dual views of the literate program as a piece of documentation and as a program should be used to define the way in which a literate program is represented and accessed. The previous two chapters have highlighted the problems inherent in subsuming either of the program or documentation structures into the other. A method of representing literate programs will be proposed in this chapter which gives equal importance to the program and documentation structure. Some possible ways of using the representation in the context of literate programming editors will be examined, and finally some of the applications of a generalised form of the representation will be briefly mentioned.

6.2 Dual-rooted grammars

Both the documentation and the program code in a literate program have hierarchical structures, and can be represented using derivation trees of their respective grammars. Attributes can be used to ascribe meaning to these derivation trees, and the semantic information derived in this way can be passed around and used in these hierarchies.

Instead of prioritising one hierarchy over the other, the structures should be merged so that they have equal priority. If we represent both the program and documentation as derivation trees of their respective grammars, there are portions of each derivation tree which correspond to each other, such as the code for a particular refinement which is described in a documented section. Figure 6–1 shows how derivation trees of the documentation grammar given in figure 6–3 and the program grammar given in figure 6–2 correspond. These portions are not identical—the documentation grammar allows refinement references in code fragments, but the program grammar does not; the expanded refinement should be derived instead. If the program grammar is augmented to associate names with sub-trees (as was done in the trial implementation in section 4.3.1) the sub-trees could be merged, and used in both contexts. A single grammar might then be used to describe the entire literate program, documentation and code, in which the derivation trees of certain productions are merged with derivation trees of related productions.

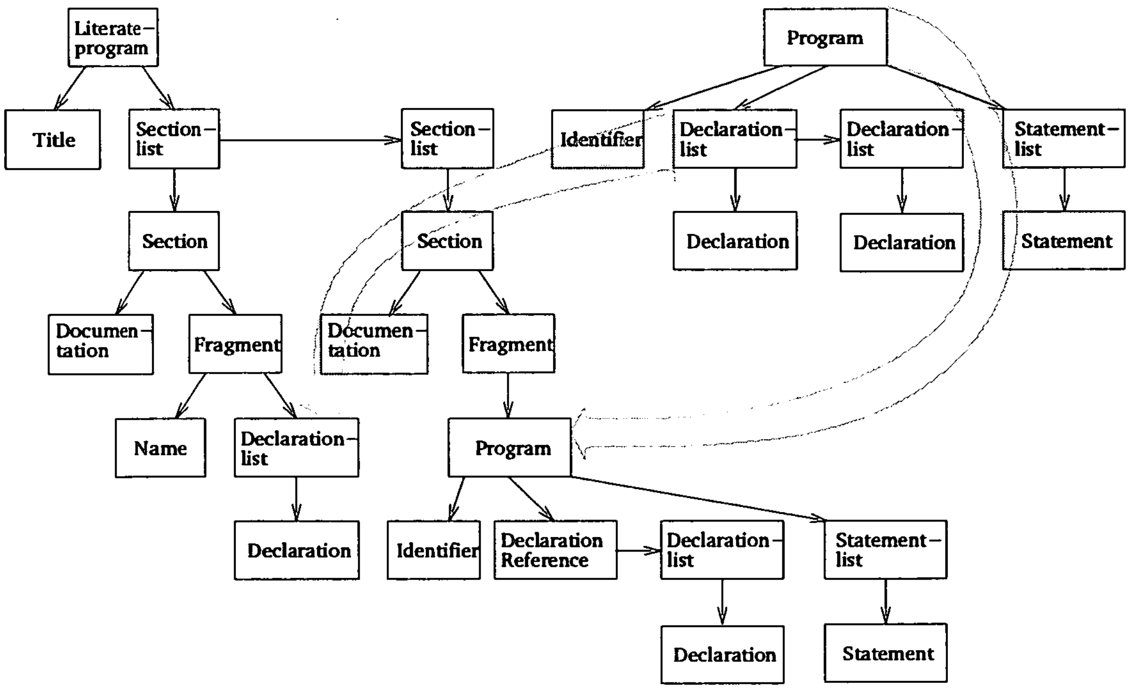


Figure 6–1: Derivation trees of literate program documentation and program showing corresponding subtrees.

```
Program ⟶
    Identifier Declaration-list Statement-list □
Declaration-list ⟶
    Declaration Declaration-list |
    □
Declaration ⟶
    Identifier Type □
Statement-list ⟶
    Statement Statement-list |
    □
Statement ⟶
    Identifier := Expression |
    if Expression then Statement else Statement |
    while Expression do Statement |
    begin Statement-list end |
    □
```

Figure 6–2: Simplified program grammar for the toy literate program editor in chapter 4.

Merging the derivation trees in this way creates an acyclic graph, rather than a tree. Each of the derivation trees of the program and documentation can still be attributed, and attributes can depend on attributes of the other derivation tree, so long as no circularities are present in the combined attribute dependency graph. Note that the attribute dependencies no longer need to be circular in order to pass information around the program and documentation trees, because the program or documentation are not being represented *as attributes* of each other. Attributes will be associated with a particular derivation tree (either program or documentation) if necessary, and may only be synthesised or inherited within that tree. The values of the attributes can be used in the other tree, so long as the constraints on cyclic dependencies are met — evaluation strategies which rely on non-circular grammars can therefore

```
Literate-program ⟶
        Title Section-list □
Section-list ⟶
        Section Section-list |
        □
Section ⟶
        Documentation Fragment |
        Documentation □
Fragment ⟶
        Program |
        Name Declaration-list |
        Name Statement-list □
Program ⟶
        as Program in figure 6–2 □
Declaration-list ⟶
        as Declaration-list in figure 6–2 □
Declaration ⟶
        as Declaration in figure 6–2 |
        Declaration-reference □
Statement-list ⟶
        as Statement-list in figure 6–2 □
Statement ⟶
        as Statement in figure 6–2 |
        Statement-reference □
```

Figure 6–3: Simplified documentation grammar for the toy literate program editor in chapter 4.

be used to evaluate the attributes.

This merged derivation tree representation for the literate program has some advantages; the syntax of the documentation and the program are both guaranteed to be correct; attribute equations can be written in a natural manner; the semantics of the language can be checked through these attribute equations.

What has just been described is a particular case of an *attributed graph.*

Attributed graphs have been described before with *attributed graph grammars* [61] and *attributed graph specifications* [2,3]. Graph grammars use graph re-writing rules to specify the derivation of a structure graph from an initial vertex. Attributed graph grammars attach attributes to vertices of the graph, which can be kept consistent by incremental evaluation of the attribute dependencies. For a large subset of the possible attributed graph grammars a *characteristic graph* can be pre-computed which describes the order in which the attribution rules of a vertex should be evaluated. Characteristic graphs can be combined during the construction of the structure graph to allow efficient evaluation of the complete attribute dependency graph.

The merged derivation trees described above for literate programs do not map onto graph grammars well. The refinement process can be modelled as inserting a reference into a sub-graph of the structure graph which corresponds to program code and inserting a code sub-graph (the refinement) into a sub-graph of the structure graph which corresponds to a documentation section. A single production which substitutes a sub-graph of the structure graph for another sub-graph cannot be used to model this process, because there are two unrelated points at which insertions are made.

The attributed graph specifications of Alpern *et. al.* [3] can be used to describe any directed acyclic graph, including graphs which are not derivable by graph grammars. Attributes are associated with edges of the graph rather than vertices, but the attribute equations are associated with vertices of the graph. Attribute equations operate directly on the values of the attributes in edges which connect vertices together. Attributed graph specifications do not provide a way of deriving structure graphs, but a way of describing and evaluating attributes of an existing structure graph.

A description which models the derivation of the whole literate program including refinements is required. The refinement process is inherently context-sensitive; it requires modifications to more than one part of the derivation graph at once, and requires the values of terminals in the documentation and program grammars to be similar; in this respect it is similar to most programming languages, which cannot be completely described by the context-free grammars used to represent their syntax. Bearing this in mind, we can describe the literate program using a context-free grammar with two start symbols, and an appropriate set of semantic functions to check the correctness of the derived strings.

DEFINITION 1 *A context-free grammar can be denoted $G = (N, T, P, S)$, where N and T are disjoint finite sets of non-terminals and terminals respectively. P is a set of productions of the form $L \rightarrow \alpha$, where L is a non-terminal and α is a string of symbols from $(N \cup T)^*$. S is a non-terminal called the start symbol.*

The *language* defined by this grammar is the set of strings of symbols derived by substituting the right-hand side of a production for instances of the left-hand side of the production until the symbols are all terminals, starting with the start symbol.

DEFINITION 2 *A context-free attribute grammar is a context-free grammar with a finite set of attributes $A(X), X \in (N \cup T)$ associated with the symbols of the grammar. $A(X)$ is partitioned into two disjoint sets, the synthesised attributes $A_S(X)$ and the inherited attributes $A_I(X)$. Each attribute a in $A(X)$ has a (possibly infinite) set of values V_a, from which one will be selected by semantic rules for each appearance of X in a derivation tree. The values of the synthesised attributes $A_S(X)$ are generated by functions of the values of $V_{A_S(X)}$ and attributes of symbols which appear in the derived strings α of productions $X \rightarrow \alpha$. The values of inherited attributes $A_I(X)$ are generated by functions of the values $V_{A_I(X)}$ and attributes of the symbol L where X appears in the string α in the production $L \rightarrow \alpha$. There are no inherited attributes of the start symbol (i.e., $A_I(S)$ is empty), and there are no synthesised attributes of terminal symbols ($A_S(t)$ is empty, where $t \in T$).*

Introducing an extra start symbol into the grammar requires changes to the formal description of the grammar. The start symbol S now becomes a pair of start symbols $S \subseteq N$, and the language of the grammar changes from being a set of strings to being a set of pairs of strings, one string derived from each start symbol. The description of an attribute grammar becomes more complex, because we have two start symbols. Two sets of attributes are required, one set related to each start symbol. Non-terminals which can be derived from either start symbol have a combined set of attributes from both sets.

Not all pairs of strings in the language of the new grammar are valid statements of literate programs. For each substring in the derivation of the documentation grammar which was derived from a production shared with the program grammar, there must be an identical substring in the derivation of the program grammar. These substrings in the derivation of the documentation grammar may wholly enclose other substrings in the same string, but may not overlap each other's boundaries. This requirement represents the refinements that can be substituted in various places in the program. The derivation trees for the program and documentation can be merged at these points. The derivation trees will not *always* be merged when the same substring is derived in both the program and documentation grammars; a substring derived from a production common to both grammars may be repeated in the program grammar derived string but not the documentation derived string. In addition, the entire string derived from the program grammar has to appear as a substring of the documentation grammar, so that the documented program can be linearised for output. Extra information is therefore needed to cross-reference the pieces of each string which correspond to each other. The semantic functions which use attributes associated with the different start symbols are restricted to these common substrings, where both attribute sets will be defined. More formally:

DEFINITION 3 *An attributed dual-rooted document grammar can be denoted $G = (N, T, P, S, A)$. N and T are disjoint finite sets of non-terminals and terminals respectively. P is a set of productions of the form $L \rightarrow \alpha$, where L is a non-terminal and α is a string of symbols from $(N \cup T)^*$. S is a pair of non-terminals (S_p, S_d) called the start symbols; the derivation strings D_p and D_d of the start symbols are strings of terminal symbols formed by repeated application of productions to non-terminal symbols appearing in the incomplete derivations of S_p and S_d. $A(X), X \in (N \cup T)$ is a finite set of attributes associated with the symbols of the* *grammar, partitioned into three disjoint sets, the synthesised attributes $A_S(X)$, the inherited attributes $A_P(X)$ derived from S_p and the inherited attributes $A_D(X)$ derived from S_d. Each attribute a in $A(X)$ has a set of values V_a, one of which will be selected for each appearance of X in a derivation tree. The values of the synthesised attributes $A_S(X)$ are generated by functions of the values of $V_{A_S(X)}$ and attributes of symbols which appear in the derived strings α of productions $X \rightarrow \alpha$. The values of the inherited attributes $A_P(X)$ and $A_D(X)$ are generated by functions of the values $V_{A_P(X)}$ or $V_{A_D(X)}$ and attributes of the symbol L where X appears in the string α in the production $L \rightarrow \alpha$ and L is derivable from S_p or S_d respectively. The predicate function F is true if the derivation string D_p appears as a substring of the derivation string D_d, and every substring of D_d generated by a production $L \rightarrow \alpha$ where L is derivable from S_p has a unique corresponding substring in D_p. The set of valid literate programs is the set of pairs of derivation strings D_p and D_d for which the function F is true. Inherited attributes in the set $A_P(X)$ where X appears in the right hand side of a production $L \rightarrow \alpha$ and L is only derivable from S_d are set to the values of the attributes of the corresponding derivation of S_p.*

The productions P include productions of both of the grammars deriving strings from S_p and S_d, so if a symbol's parent non-terminal was derivable from both S_p and S_d, the symbol is also derivable from S_p and S_d.

Note that this definition could have been given using an alternative formalism such as graph grammars, but the constraints on valid literate programs are easier to specify as predicates on strings.

6.3 Structured editing with dual-rooted grammars

The new representation proposed in the previous section should make it easier to manipulate literate programs. We have to consider how a structure editor which uses this representation will appear to the user, and how it will maintain the constraints on matching subtrees.

Since there are two roots and two derivation trees in the representation, the editor should either present two windows, one with each tree, or allow switching between the trees shown in the current window. Different view specifications can be used to compress refinement references in the displayed documentation tree, and remove refinement names from the program tree.

A difficult point with the derivation trees for literate programs is that the derivation of the top-level program must appear within the derivation tree of the documentation. There are two possible solutions to this problem; we can either require a derivation for the program to be present in the documentation at all times, or develop the derivations separately and allow the program derivation to be inserted into the documentation tree when the user desires.

With the first approach, placeholders have to be inserted into the initial derivation tree of the documentation for all of the ancestor nodes of the program subtree. There may be an infinite number of possible derivation trees which derive the program subtree, so it is assumed that the shortest derivation would be used. These ancestor nodes have to be provided at some point during development of the program anyway, so providing a template for the derivation does not detract from the expression of the literate program. There is a problem with requiring the program subtree to be present: the program subtree cannot be deleted, even temporarily. This means that the common cut and paste idiom for editing cannot be used to move the program subtree around, unless it is implemented as an indivisible editing operation.

With the second method, cutting and pasting the program derivation into the appropriate place in the documentation could be performed, or a method of linking the program view to the documentation view could be created (such as saving the position and extent of structural selections, and linking the derivation tree with the position of the current selection).

The problem of how to place the program subtree within the documentation subtree is a case of the more general problem of how to link nodes in the program and documentation, and ensure that their attributes remain consistent. The solution to this problem will define how the literate program editor can be used to develop programs. If the attributes of both derivation trees and the merged subtrees are required to be consistent at all times, the fragments of program derived from documentation nodes must be linked into the program tree at all times (thus complying with the constraints on valid derivations in the previous section). This constraint might be implementable by using a transformation on the documentation node enclosing the program subtree, which substitutes a portion of the program subtree with a refinement and adds a new section containing the linked refinement. Unfortunately, this would require an extension to the sort of transformations possible in the Synthesizer Generator, because the place at which the refinement is required may be arbitrarily deep in the program derivation tree. A more serious objection to this method of implementation is that it forces the user into top-down design of the program. This denies one of the fundamental tenets of literate programming, that the program can be developed and expressed in the order most suitable to the programmer.

The alternative is to allow the derivations to be developed separately, but to attribute program fragments when they are linked into the program tree. If program fragments which are not part of the program tree are attributed, there may be problems with missing inherited attributes from the program tree. The requirement for turning the attribution on or off in these cases is that the subtree should be derived from a production (via either derivation tree) which provides all of the inherited attributes required. Similarly, if a subtree synthesises attributes which are used by its parent in one or more derivations, but the subtree has attribution turned off, the attribution of the parent should be turned off. Turning off attribution can be made a bit more selective by examining the attribute dependency graphs, and not evaluating any attribute dependency chains which have missing attributes in them. Upward remote attribute references can only be computed for attributes if they are associated with a particular derivation tree, and hence a particular root node.

The actual implementation of an editor based on the new representation can take two approaches; the derivation trees for both of the grammars can be stored as trees, with updates to linked nodes copied to the other tree, or the derivation trees can be merged into an acyclic graph. The latter approach is probably simpler, because updates to attributes during attribute evaluation do not need to be copied between the trees. The implementation of a parsing phase for input files raises some points; two input syntaxes for the program and documentation grammars need to be parsed in parallel, and the linkages between them created. To simplify the parsing phase, a new root node can be introduced which has a single production deriving the two original root nodes. Linking of the derivation trees can then be done by a semantic function that traverses the documentation tree searching for program fragments and their corresponding program tree derivation. The correspondence is defined by the names of the refinements used, rather than the structure of the subtrees, so that subtrees in the program derivation which are coincidentally the same as documented program fragments do not get linked to the program fragments.

The Synthesizer Generator has a type of phylum called a PTR which allows references to SSL values. Unfortunately, this mechanism cannot be used to implement an editor with dual derivation trees because PTRs to other phyla may not appear in productions. Even with the higher-order attributes introduced in Synthesizer Generator release 3.5, the references would still not be editable.

6.4 Many-rooted grammars

The previous sections have shown how representing a literate program as interlinked derivation trees of a grammar with two start symbols could be used to provide a more natural editing interface than can currently be constructed using editors created by the Synthesizer Generator. There are two generalisations to this model that could be made, which expand its potential application. The number of start symbols could be increased beyond two, and more than one derivation could be made from each start symbol.

Allowing than two start symbols in the grammar would allow other views of a document to be represented naturally; for instance, a specification for the program might be interleaved with the program and documentation, using a specification language such as OBJ or Z. More than one derivation from each start symbol would allow multiple programs per document, documentation in more than one language, and multiple versions of the documentation (for example, full program documentation and a short paper on an interesting algorithm could be combined, or a revision history could be maintained as separate documents).

The formal description of the grammar with multiple start symbols is similar to the description of the grammar with two start symbols:

DEFINITION 4 *An attributed many-rooted document grammar can be denoted $G = (N, T, P, S, A)$. N, T and P have the same meanings as in a dual-rooted document grammar. S is a set of start symbols such that $S \subseteq N$. A is a set of attributes divided into the disjoint partitions for the synthesised attributes $A_S(X)$ and the inherited attribute partitions $A_x(X)$, $x \in S$. The language of the grammar is a set of strings, of which a subset of valid strings will be determined by a predicate function F. The predicate function will depend upon the languages represented by each start symbol in S.*

The particular semantic restriction present in the literate program grammars examined (that the root of the program tree must appear in the documentation tree) does not need to hold for all grammars. There in now a set of attributes associated with each start symbol. Non-terminals which can be derived from more than one start symbol have combined sets of attributes from all of the attribute sets associated with the start symbols.

Extending the formal description to allow multiple derivation trees from each start symbol can be done by specifying an *actual* set of start symbols S_a, whose members are taken from a *possible* set of start symbols S_p, $S_p \subseteq N$. Members of S_p may appear more than once in S_a. This extension would have be implemented by allowing separate development of the derivation trees, and attribution of the nodes as and when the sub-trees are linked into positions which provide their required inherited attributes. The facility to introduce new derivation trees of a particular start symbol is required, as the author of a document may not anticipate all of the possible versions of the final document which might be created.

7 Conclusions

7.1 Summary

This thesis has examined how the literate programming methodology can be facilitated by structured editing tools.

- In chapter 1 the literate programming concept for documenting programs was introduced, and how it differs from program pretty-printing was presented. A brief introduction to syntax-directed editing was given.

- The development of literate programming systems from Knuth's WEB to the current day was traced in chapter 2. A set of criterion by which a programming system can be judged literate were presented. Many of the systems examined were lacking in one or more of these criterion. Some of the related work in program comprehension was also highlighted. An overview of some syntax-directed editing systems was presented.

- In chapter 3 some of the problems with current literate programming systems were raised, and an overview of how syntax-directed editing might help solve these problems and assist literate programming was presented.

- Chapter 4 described the design and implementation of two small literate programming editors for a 'toy' programming language using the Synthesizer Generator. The compromises which were made in mapping the program structure onto the documentation structure and vice versa were shown. A larger-scale implementation of a literate Pascal editor was described, which showed problems with the automatic detection of semantic errors.

- The causes of the problems in detecting semantic errors in the literate Pascal editor were examined in chapter 5. It was shown that cyclic attribute dependencies are a consequence of the mapping the program structure onto the documentation structure or vice versa. The dependencies generated for the literate program editors were shown to be convergent, and it was shown how they could be evaluated, but the performance of the evaluation method was shown to be unacceptable for interactive editing.

- In chapter 6 a representation for literate programs was proposed, using grammars with multiple start symbols and shared derivation sub-trees. This representation avoided the circular attribute dependencies which caused problems with the implementations in chapter 4, and thus would be better for interactive editors. Some issues in using this representation to implement literate program editors were raised. A generalisation of the representation to allow many start symbols and derivation trees was briefly presented.

7.2 Future Work

There are several parts of the work presented in this thesis which may be worth developing further.

The technique of finding cyclic paths through a dependency graph used in chapter 5 may be applicable to other circular attribute evaluation algorithms, especially where attributes can take list or array values. Further work on deciding when this method may be useful and applying it to general circular attribute evaluation algorithms is required.

The definitions of dual-rooted and many-rooted grammars described in chapter 6 need careful scrutiny. Algorithms for ordering the attribute evaluations in these grammars need to be presented. It is obvious that there are no circular attribute dependencies introduced by the representation itself (the program and documentation attribute dependencies follow the program and documentation derivation trees, which are both sub-trees of the merged derivation graph). Algorithms for ordering attribute evaluations in these grammars may be based upon the Reps's incremental attribute evaluator [96], as the basic tree structure is still present in most of the attribute dependencies. An alternative approach to evaluating the attribute dependencies would be to demonstrate a method to transform the derivation trees into attributed graph specifications.

If an editor is to be built using these grammars, a more general method of specifying the valid derivations of the grammar may also be necessary. This would make it possible to build an editor-generating system which does not have special knowledge about the particular structure of literate programs built with it. The method of specifying valid derivations might be to predicate acceptance of user changes on attribute evaluation; if an alteration causes a predicate attribute to evaluate to the false value, the alteration is not permitted and the attributes are restored to their original values. This would provide a general mechanism for enforcing context-sensitive syntax or static semantic correctness. This mechanism would probably not be used for all semantic checking, because it would make writing and changing programs awkward if all declarations and types had to be correct all of the time.

An editor based on these methods would provide a testbed to check their validity, but there is a piece of implementation work which could be performed without such an editor. This is to extend the editor in chapter 4 to include publication tools, such as a table of contents, index of identifiers, module cross-reference and WEB program importer. This could be used to study the human interaction aspects of using the literate program editor. Brown [23] showed some subjective evidence that a literate program browser improved program comprehension when performing typical maintenance tasks. The literate program editor, albeit without full interactive semantic checking, could be used to examine whether better programs are *developed* using a literate program editor, and whether the refinement placements and views chosen are useful.

7.3 Conclusions

Both literate programming and syntax-directed editing can improve a programmer's working environment. Literate programming allows the combination of design methods to suit the program and programmer, and makes documentation of design decisions an integral part of the program. The benefits of literate programming are mainly seen at the program maintenance stage, when the increase in comprehensibility over conventional programs has an effect. Syntax directed editing gives the programmer guidance in the process of implementing the program. Restructuring operations are easier, and syntactic and semantic errors can be noted and avoided, reducing the number of edit-compile cycles to create the working program.

The combination of syntax-directed editing and literate programming could provide editors which can assist in creating the documentation and code of programs.

The attribute grammar representation of programs used in editors created by the Synthesizer Generator is a natural and powerful way of representing conventional programs, but it does not cope with literate programs in an acceptable way. In order to retain the power of attribute grammars when editing literate programs, we need to extend the attribute grammar to represent both the documentation and program structures explicitly. This can be done by introducing extra start symbols into the grammar and merging the derivations of the start symbols into an acyclic graph.

The generalisation of the attribute grammar to include multiple start symbols and multiple derivations per start symbol has many possible applications in representing non-linear documents.

There is more fundamental work to be done to realise easy creation of fully structure-based literate programming editors. The method of describing the structure of literate programs presented in this thesis may make automatic generation of literate program and web-structured information easier.

Bibliography

[1] Paul W. Abrahams. "Typographical Extensions for Programming Languages: Breaking out of the ASCII Straitjacket". *ACM SIGPLAN Notices*, Vol. 28, No. 2, pp. 61–68, February 1993.

[2] Bowen Alpern, Alan Carle, Barry Rosen, Peter Sweeney, and Kenneth Zadeck. *Incremental evaluation of attributed graphs*. Technical Report CS-87-29, Department of Computer Science, Brown University, December 1987.

[3] Bowen Alpern, Alan Carle, Barry Rosen, Peter Sweeney, and Kenneth Zadeck. "Graph Attribution as a Specification Paradigm". *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pp. 120–129, Boston, Mass., November 1988. Association for Computing Machinery. *ACM Sigplan Notices 24*, 2.

[4] N. Anand. "Clarify Function!". *ACM SIGPLAN Notices*, Vol. 23, No. 6, pp. 69–79, June 1988.

[5] W. Appelt and K. Horn. "Multiple changefiles in WEB". *TUGboat*, Vol. 7, No. 1, p. 20, March 1986.

[6] Mouloud Arab. "Enhancing Program Comprehension: FORMATTING and DOCUMENTING". *ACM SIGPLAN Notices*, Vol. 27, No. 2, pp. 37–46, February 1992.

[7] A. Avenarius, S. Oppermann, I. Peides, and J. Stritzinger. *The FWEB System of Structured Documentation*. Report PI-R6/89, Institut für Praktische Informatik, Technische Hochschule Darmstadt, 1989.

[8] Adrian Avenarius and Siegfried Opperman. "FWEB: A Literate Programming System for Fortran8x". *ACM SIGPLAN Notices*, Vol. 25, No. 1, pp. 52–58, January 1990.

[9] Ronald Baecker and Aaron Marcus. "Design Principles for the Enhanced Presentation of Computer Program Source Text". *Proceedings CHI'86 (Human Factors in Computing Systems)*, pp. 51–58, New York, NY, April 1986. Association for Computing Machinery.

[10] Ronald Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs*. Addison-Wesley, Reading, MA, 1990. ISBN 0-201-10745-7.

[11] R. Bahlke and G. Snelting. "The PSG system: From formal language definitions to interactive programming environments". *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 4, pp. 547–576, October 1986.

[12] Kent Beck and Ward Cunningham. *The Literate Program Browser*. Technical Report CR-8-52, Computer Research Laboratory, Textronic, Incorporated, 1986.

[13] Mordechai Ben-Ari. "Foreet: A tool for design and documentation of Fortran programs". *Software – Practice and Experience*, Vol. 16, No. 10, pp. 915–924, October 1986.

[14] Jon L. Bentley. "Programming Pearls—Literate Programming". *Communications of the ACM*, Vol. 29, No. 5, 1986.

[15] Jon L. Bentley and Donald E. Knuth. "Programming Pearls". *Communications of the ACM*, Vol. 29, No. 6, 1986.

[16] Judy M. Bishop and Kevin M. Gregson. "Literate Programming and the LIPED Environment". *Structured Programming*, No. 1, pp. 23–34, 1992.

[17] Peter Breitenlohner. "Still another aspect of multiple change files: The PATCH processor". *TUGboat*, Vol. 9, No. 1, pp. 11–12, April 1988.

[18] Preston Briggs. *NuWeb*. Announcement to LITPROG@edu.SHSU mailing list, April 1993.

[19] Frederick P. Brooks Jr. *The mythical man-month*. Addison-Wesley, 1982.

[20] M. Brown and D. Cordes. "A literate programming design language". *COMPEURO'90, Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering*, pp. 548–549, Tel Aviv, Israel, May 1990.

[21] M. Brown and B. Czejdo. "A Hypertext for Literate Programming". Selim G. Akl et. al. (editor), *Advances in Computing and Information, Proceedings of International Conference ICCI'90*, pp. 250–259, Niagara Falls, Ontario, Canada, May 1990. Lecture Notes in Computer Science 468.

[22] Marcus Brown and David Cordes. "Literate programming applied to conventional software design". *Structured Programming*, No. 11, pp. 85–98, 1990.

[23] Marcus E. Brown. *An Interactive Environment for Literate Programming*. PhD thesis, Texas A&M University, College Station, TX, August 1988.

[24] Marcus E. Brown. *The Literate Programming Tool*. Technical report, Computer Science Department, Texas A&M University, August 1988.

[25] Marcus E. Brown and Bart Childs. *An Interactive Tool for Literate Programming*, (unpublished). in *Third Workshop on Empirical Studies of Programmers*, unpublished proceedings, Austin, Texas, April 1989.

[26] Marcus E. Brown and Bart Childs. "An Interactive Environment for Liter-

ate Programming". *Structured Programming*, Vol. 11, No. 1, pp. 11–25, 1990.

[27] Carlos F. Bunge and Gerardo Cisneros. "Modular libraries and literate programming in software for *ab initio* atomic and molecular electronic structure calculations". *Computers & Chemistry*, No. 12, pp. 85–89, 1988.

[28] Bart Childs. "Literate programming, a practitioner's view". *TUGboat*, Vol. 13, No. 3, pp. 261–268, 1992.

[29] Bart Childs and Timothy Jay McGuire. "Symbolic computing, automatic programming, and literate programming". *Computational Techniques and Applications, Proceedings of CTAC-91*, Adelaide, Australia, July 1991. Australian National University.

[30] David Cordes and Marcus Brown. "The literate-programming paradigm". *Computer*, Vol. 24, No. 6, pp. 52–61, June 1991.

[31] Ward Cunningham and Kent Beck. *Scroll Controller Explained, An Example of Literate Programming in Smalltalk*. Technical Report CR-86-53, Computer Research Laboratory, Textronix Incorporated, 1986.

[32] R. M. Damerell. "Error detecting changes to Tangle". *TUGboat*, Vol. 1, No. 7, pp. 22–24, 1986.

[33] Peter J. Denning. "Announcing Literate Programming". *Communications of the ACM*, Vol. 30, No. 7, p. 593, July 1987.

[34] Christine Detig and Joachim Schrod. *MWEB, A WEB system for Modula-2*. Report PI-R10/88, Institut für Praktische Informatik, Technische Hochschule Darmstadt, March 1988.

[35] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. *Interactive Programming Environments*, "Programming environments based on structured editors: The MENTOR experience", pp. 128–140. McGraw-Hill, New York, NY, 1984.

[36] V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Mélèse. "Document structure and modularity in Mentor". Peter Henderson (editor), *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pp. 141–147, Pittsburgh, Pennsylvania, April 1984. Association for Computing Machinery.

[37] Angus Duggan. *Literate Programming: A Review*. LFCS Report ECS-LFCS-93-263, Department of Computer Science, The University of Edinburgh, April 1993.

[38] G. Engels, M. Nagl, and W. Schaefer. "On the Structure of Structure-Oriented Editors for Different Applications". *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pp. 190–198. Association for Computing Machinery, 1986. *ACM Sigplan Notices 22*.

[39] G. Engels and W. Schaefer. *Formal methods and software development : Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 186 of *Lecture Notes in Computer Science*, "Graph Grammar Engineering: A Method Used for the Development of an Integrated Programming Support Environment", pp. 179–193. Springer-Verlag, 1985.

[40] Scott E. Fahlman. *The Igor Project*. Announcement to Usenet newsgroup `comp.lang.dylan`, August 1993.

[41] Rodney Farrow. "Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars". *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 85–98, Palo Alto, California, June 1986. Association for Computing Machinery.

[42] S. Feiner, S. Nagy, and A. van Dam. "An experimental system for creating and presenting interactive graphical documents". *ACM Transactions on Graphics*, No. 1, pp. 58–77, 1982.

[43] James C. Ferrans, David W. Hurst, Michael A. Sennett, Burton M. Covnot, Wenguang Ji, Peter Kajka, and Wei Ouyang. "HyperWeb: a Framework for Hypermedia-Based Environments". Herbert Weber (editor), *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pp. 1–10, Tyson's Corner, Virginia, USA, December 1992. Association for Computing Machinery.

[44] C. N. Fischer, Gregory F. Johnson, Jon Mauney, Anil Pal, and Daniel Stock. "The Poe Language-Based Editor Project". Peter Henderson (editor), *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pp. 21–29, Pittsburgh, Pennsylvania, April 1984. Association for Computing Machinery.

[45] Jim Fox. "Webless literate programming". *TUGboat*, Vol. 11, No. 4, pp. 511–513, 1990.

[46] David B. Garlan and Philip L. Miller. "GNOME: An Introductory Programming Environment Based on a Family of Structure Editors". Peter Henderson (editor), *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pp. 65–72, Pittsburgh, Pennsylvania, April 1984. Association for Computing Machinery.

[47] Peter Gragert and Marcel Roelofs. *Reduce WEB version 3.4*, (unpublished). Available by anonymous ftp from `utmf0.math.utwente.nl`.

[48] Klaus Guntermann and Wolfgang Rülling. "Another approach to multiple changefiles". *TUGboat*, Vol. 7, No. 3, p. 134, October 1986.

[49] Klaus Guntermann and Joachim Schrod. "WEB adapted to C". *TUGboat*, Vol. 7, No. 3, pp. 134–137, October 1986.

[50] Klaus Guntermann and Helmut Waldschmidt. "Modular programming with WEB". *Electrosoft*, Vol. 1, No. 1, pp. 27–43, March 1990.

[51] Wilfred J. Hansen. *Interactive Programming Environments*, "Program-