Editable Software Views

by

Michael Thomas Hardy

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 1995

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Understanding software is a very expensive component of software development and maintenance. As a result, tools that assist in the process of software understanding play an important part in reducing software development and maintenance costs. As part of this research, the current literature was examined and it was determined, surprisingly, that very little work had been done in developing tools to assist programmers in understanding source code. Many of the tools presented in the literature suffer from one or more drawbacks that limit their usefulness for a software developer or maintainer. To address some of these limitations, an extensible prototype system that presents extrinsic information about a program in editable source code views was designed and developed. The views are created by embellishing the source code using changes in font family, size, and style, text colour, and text elision. This thesis presents the design and implementation of this system. This research represents a step in the development of tools to assist in software understanding, but it is clear that more effort must be directed towards this area of research.

Acknowledgments

I would like to thank my supervisor, Dr. Richard Bartels, for his encouragement and enthusiasm in my thesis research. His comments and suggestions were invaluable in the writing of my thesis.

Special thanks go to my wife Mary Anne for her constant support and encouragement throughout my course and thesis work. Her warm heart and kindness kept me sane during many crazy times. I would also like to thank my parents for instilling in me the thirst for knowledge and for supporting me throughout my education.

I would also like to thank the Natural Sciences and Engineering Research Council, the Institute for Computer Research, the University of Waterloo, and the Department of Computer Science of the University of Waterloo for helping fund this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software developers and maintainers use a large variety of tools to assist them in their daily work. These tools include compilers, profilers, browsers, test coverage tools, and version control systems. Many of these tools assist the programmer by extracting and presenting information about the software being developed or maintained. This information can help the programmer develop and maintain programs by identifying errors, sections of source code that are bottlenecks, areas of the program that have not been tested, etc. One of the problems with all of these tools is that each presents information in a different format. Combined with the volume of information presented, this can often make it difficult for the programmer to locate useful information. Another problem with most of the tools is that the information they provide cannot be viewed in conjunction with the source code of the program. Tools that support this functionality do not allow the editing of the source code while the additional information is being viewed. Both of these

restrictions prevent the software developer or maintainer from making modifications to the program immediately based on the information from these external tools.

This thesis attempts to address these problems by developing a prototype software development and maintenance system that is both customizable and extensible. This system allows software developers and maintainers to use external tools to extract information from programs and to display this information in conjunction with the source code for the program. The information is presented to the user by embellishing the source code using colour and font changes in addition to text elision[1]. The programmer can also modify the source code while viewing the combination of external information and source code. These editable views of source code and auxiliary data will hopefully improve the understandability of source code and contribute to an improvement in software developer and maintainer productivity.

In Chapter 2, literature related to this thesis is reviewed. A discussion of software maintenance costs and of the issue of program understanding is provided. Tools attempting to address this issue are presented. This chapter concludes with a justification of the research and a brief description of the system that was developed

Chapter 3 describes the design and implementation of the various components of the system.

---

[1]Text elision is the process of hiding or omitting text.

We show an example of how the system can be used in Chapter 4.

In Chapter 5, the results of this thesis work are discussed. The thesis concludes with future research directions.

The system makes use of a tagging language and two special data files. These are described in the appendices.

Chapter 2

Literature Review

This chapter discusses the issues motivating this thesis. Section 2.1 reviews literature dealing with software maintenance costs, the problems associated with understanding software, and tools attempting to address these issues. Based on this literature review, justification for the thesis work is given in Section 2.2. Section 2.3 concludes this chapter with a brief overview of the prototype system that was developed.

2.1 Previous Work

"Software maintenance accounts for between 50 and 90 percent of the total life cycle expenditures on [a] programming system" [CLEVE89, p. 24]. This view is shared by Corbi [CORBI89] and Robson *et al.* [ROBSO91]. Due to the high cost of software maintenance, tools and techniques that can reduce the time spent on software maintenance tasks are in high demand. To reduce the costs of software maintenance, it is necessary to

determine what causes them to be high. Oman [OMAN90b] and Robson *et al.* [ROBSO91] believe that the process of software comprehension is the most critical aspect of software maintenance. This opinion is shared by many including Cleveland [CLEVE89], Chen and Ramamoorthy [CHENY86], and Corbi [CORBI89], who claim that understanding an existing software system is the most time-consuming aspect of software maintenance, involving anywhere from fifty to ninety percent of a maintenance programmer's time. Clearly, tools that can help programmers understand software are an important part of reducing software development and maintenance costs. In his review of eight code visualization tools, Oman comments that "several prominent researchers have suggested that tools and techniques for understanding large programs will be one of the major challenges of the 1990s" [OMAN90b, p. 59]. This view is shared by Corbi who feels that the development of software tools will move away from the production of applications that help develop new programming systems and towards those that will assist in program understanding and enhancement [CORBI90].

In order to assist in program understanding, it is necessary to present information about the program to the user. A variety of techniques have been developed for presenting the intrinsics of the program (e.g. syntactic structure, loop or function call structure, etc.). These techniques can assist developers in understanding the source code independent of extra information. Oman and Cook determined that typographic style (source code formatting and commenting) significantly impacts program comprehension [OMAN90a]. They found that style can provide visual cues to the underlying structure of the code. Love did similar work investigating the effects of paragraphing and control flow but found that neither had any significant effect on program understanding [LOVET77].

```
Pixmap ReadBitmapFile(d, filename, width, height, x_hot, y_hot)
        Drawable d;
        char *filename;
        int *width, *height, *x_hot, *y_hot;
{
        Pixmap bitmap;
        int status;

        status = XReadBitmapFile(dpy, RootWindow(dpy, screen), filename,
                width, height, &bitmap, x_hot, y_hot);

        if (status != BitmapSuccess) bitmap_error(status, filename);

        return(bitmap);

} /*end ReadBitmapFile */
```

Figure 2-1: "Pretty Printer" Formatting

Indentation of source code also appears to have an effect on the program's comprehensibility [MIARA83]. These observations have led to the development of many "pretty printer" programs designed to format source code in a way that will improve the readability and, as a result, the understandability of the program. Figure 2-1 [OMAN90a, p. 510] shows an example of a program formatted using one of these tools.

Other approaches have been developed that are suitable for presenting both intrinsic and extrinsic information. Extrinsic information is derived from processing the program using external tools such as compilers, profilers, version control systems, etc. to extract additional information about the program. Colour has been shown in many situations to be beneficial for increasing attention to a detail [BENBA86, JONES62, DESAN84]. Colour seems to be effective in improving decision making, but it is unclear whether or not it aids in the comprehension of information. Tapp's work suggests that embellishing source code using colour helps programmers understand the code better than when no embellishments are used [TAPPR94]. The use of fisheye views is another technique that has been tried to improve the presentation of information. Fisheye views present current or high interest information in full detail, and information that is not as important in successively less detail. Figure 2-2 [FURNA86, p. 21] shows a sample fisheye view of some source code. The "..." indicate where lines of the source code have been hidden. The effect of fisheye views is to present information in addition to the context of the information. This is important because "people naturally perceive the world using both

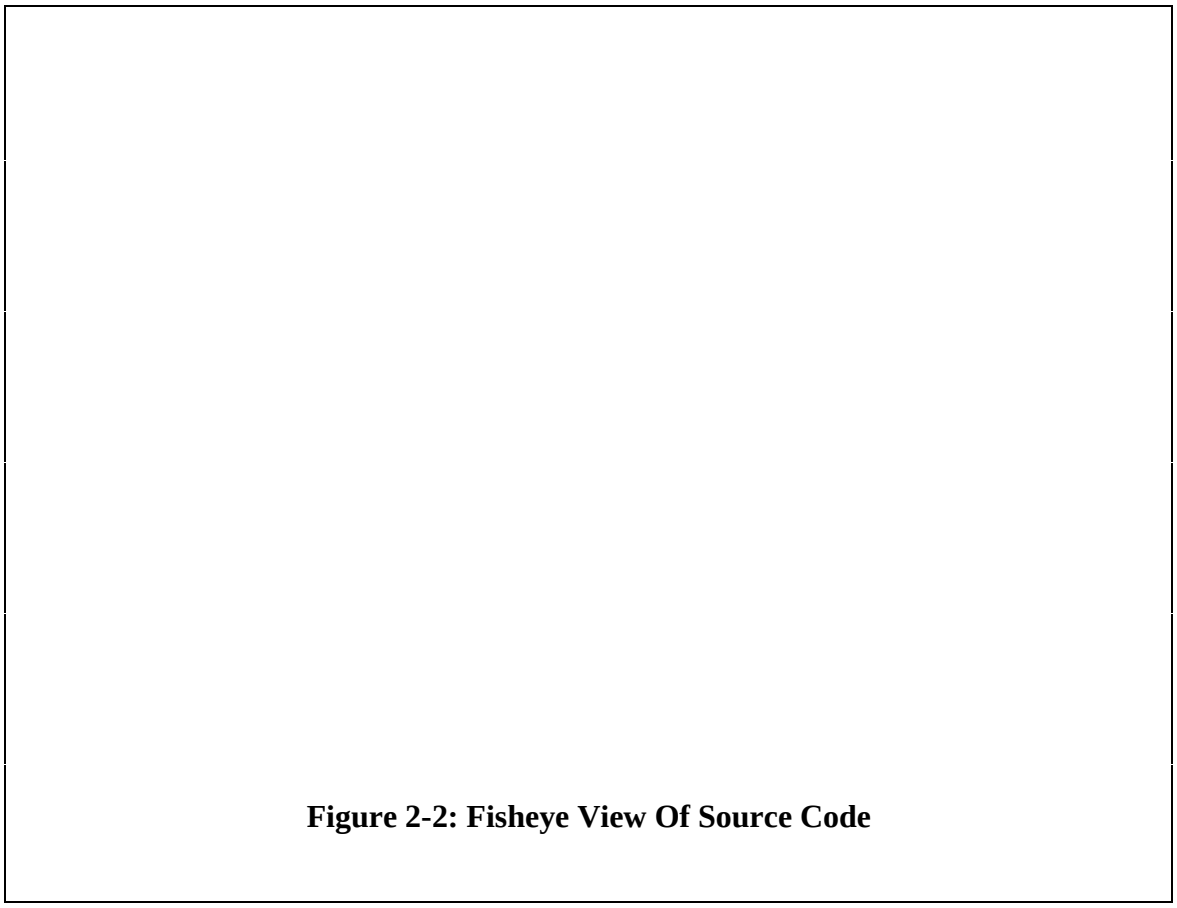

Figure 2-2: Fisheye View Of Source Code

local detail and global context" [SCHAF93, p. 87]. Schaffer *et al.* did work on the effectiveness of fisheye views in navigating information networks and found that tasks were completed faster when fisheye views were used. It is possible that this technique could help programmers better understand source code.

Reading the source code is essential to understanding the details of any program but it can be difficult to extract the relevant information from the program because of the large volume of information contained in the source code. Techniques to assist in program comprehension that are not based on reading code have also been investigated.

Representing program structure and control flow pictorially is an approach examined by Roman and Cox in 1993. They comment that these pictorial representations can simplify and enhance the explanation of specific aspects of a program [ROMAN93]. Brown and Sedgewick contend that "it is possible to expose the fundamental characteristics of a broad variety of programs through the use of dynamic (real time) graphic displays" [BROWN84, p. 177]. They believe that algorithm animation can be of significant benefit in several contexts.

In addition to techniques for presenting information, a large number of tools have been

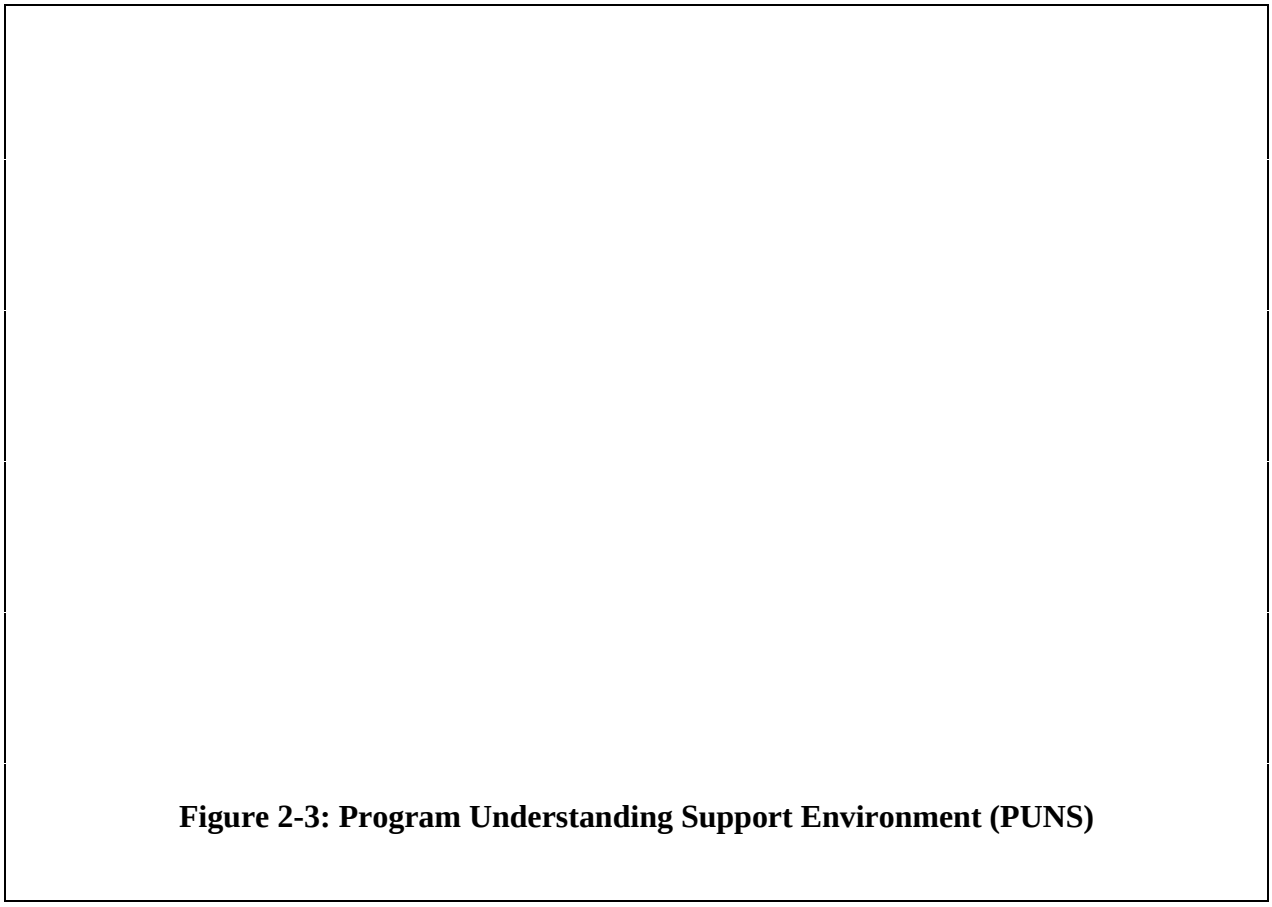

Figure 2-3: Program Understanding Support Environment (PUNS)

designed to assist the software developer and maintainer. Many of these tools extract information from the programs that the programmer can view. Examples of this type of tool include test coverage tools (e.g. tcov), profilers (e.g. prof), and parsers. Another example is the C Information Extractor (CIA) [CHENY86] which uses static analysis techniques to extract relational information from C programs and stores this information in a database for future access.

While these information extraction tools can be useful, it can be difficult to navigate through the large quantity of information that they generate. One way to deal with this volume of information is to use tools that provide program visualizations. Roman and Cox define a program visualization as "a mapping or transformation of a program to a graphical representation" [ROMAN93, p. 12]. A number of applications that support different forms of program visualizations have been developed. The Maintenance Engineering Environment (ME2) is a prototype system that maintains a database of information (syntactic, control flow, data flow, etc.) about a program. This database is accessed by tools that provide visualizations of the information such as a graphical representation of the module-calling hierarchy. A similar approach is used by the Program UNderstanding and Support (PUNS) environment developed by Cleveland [CLEVE89] (Figure 2-3 [CLEVE89, p. 333]). PUNS uses static analysis to detect the low-level relationships that exist within the program and then consolidates this information into a repository. The information in this repository can then be presented in a number of ways

(call graphs, control flow graphs, etc.). Eick, Steffen, and Sumner developed a unique system for visualizing software statistics called Seesoft [EICKS92]. Line-oriented data from different sources (version control systems, static analyses, dynamic analyses) are analyzed and presented by reducing files so that each line of code is mapped to a thin line on the display. Each line is then coloured based on the statistic being presented (Figure 2-4 [EICKS92, p. 958]).

Many maintenance and development tools can be classified as browsers. Tools such as the ET++ browser [WEINA89], the EDSA browser [VANEK89], the Visual System Browser [HUDLI89] and the browser developed by Sametinger [SAMET90] provide static, non-editable views of the source code. These views can include object or class definitions as in the ET++ and Sametinger browsers, control or data flow diagrams as in the EDSA browser, procedure call trees as in the Visual System browser, and more. A typical browser is shown in Figure 2-5 [WEINA89, p. 79]. Traditional browsers have two major drawbacks, the first being that most provide only one view of the source code. If the user requires a different view than that provided by the browser, a different browser must be used. Some tools have been created which support multiple source code views. Figure 2-6 shows PegaSys [MORIC85, p.150] which is an Ada development system that uses pictures in a number of ways to convey information about the software, including information represented in flow charts, structure charts, data flow diagrams, and module interconnection languages. Seela [HARBA90] is another multiple view source code

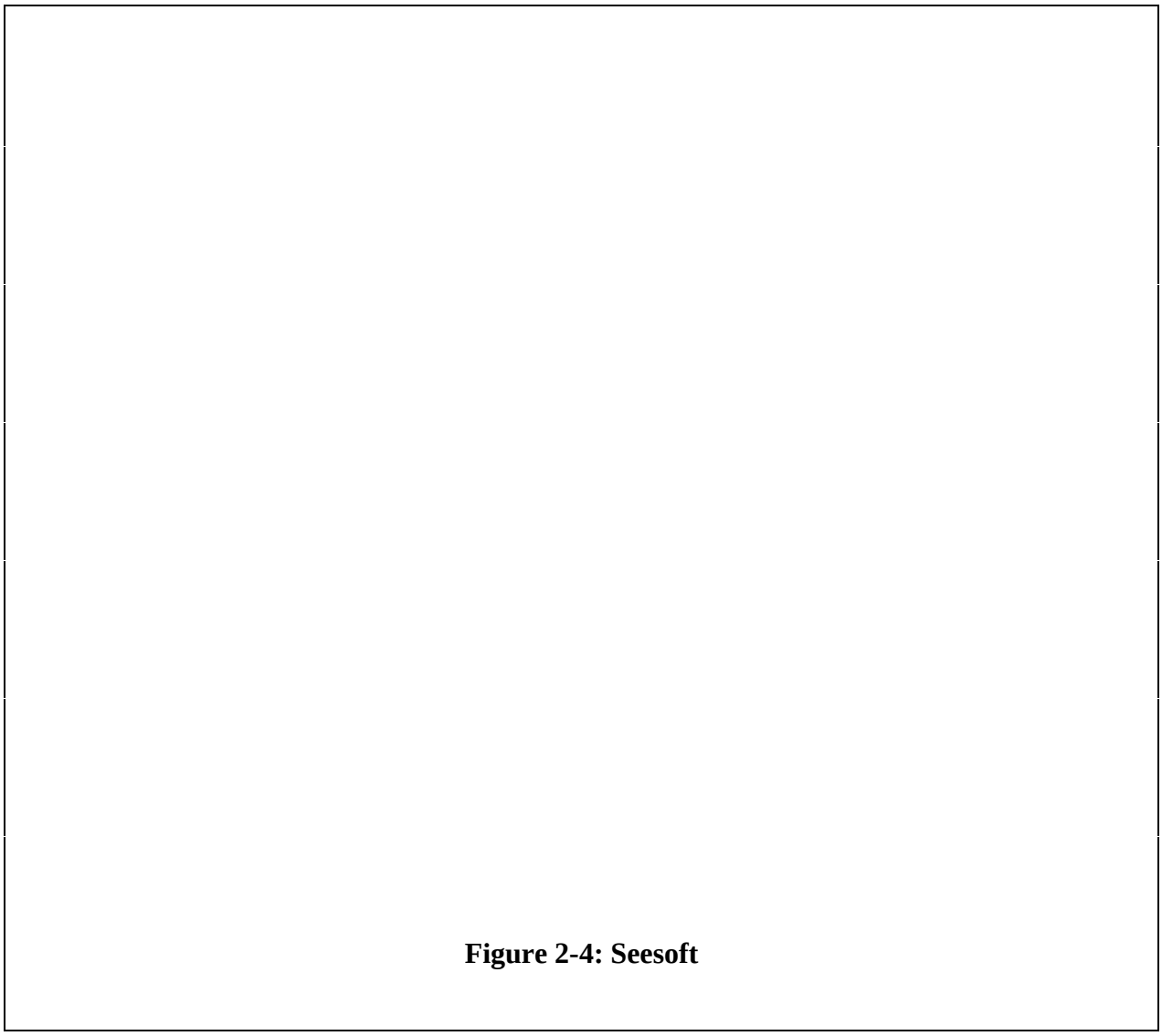

Figure 2-4: Seesoft

browser. It was designed as an interactive reverse-engineering tool for Ada, COBOL, C, Pascal, PL/M or FORTRAN programs and provides a structured editor, browser, pretty printer, and source code document generator. These tools, while more flexible than traditional browsers, are still affected by the second drawback that limits the usefulness of browsers which is that the provided views are not editable. After using a browser or

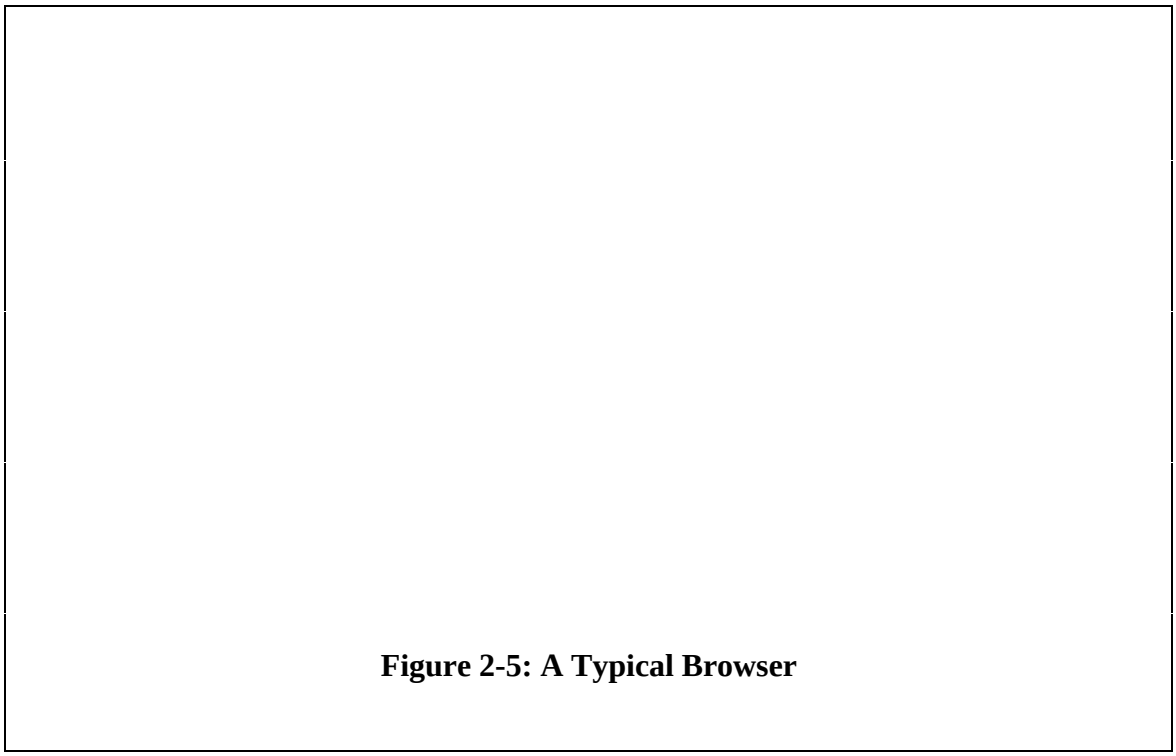

Figure 2-5: A Typical Browser

similar “read-only” tool to view information about a program, the user must “switch modes” to another application to modify the program.

A few applications have been developed that attempt to address the problem of editable views. Software Maintenance, Analysis, and Re-engineering Tools (SMART)System [EVANS90] contains a configurable text- and language-sensitive editor that allows the user to view source code as text or graphics, format the view of the source code to suit a particular style, and to apply different filtering techniques to control the level of detail shown. SMARTSystem also provides call-graphs to assist in program comprehension. In his 1985 paper, Reiss discusses the PECAN family of program development systems [REISS85]. PECAN is an extensible programming environment that

provides multiple views of the program, its semantics, and its execution (Figure 2-7 [REISS85, p. 284]). Currently PECAN provides a syntax-directed editor, and a flow chart as the supported views. Hypertext is integrated in the HyperCase system along with a knowledge-based document repository. This system provides a "visual, integrated, and customizable software engineering environment consisting of loosely coupled tools for presentations involving both text and graphics." [CYBUL92, p. 83] Cordy, Eliot, and Robertson use a unique technique to convey information in the Turing Tool [CORDY90]. The Turing Tool creates all of its editable views using source code elision, which is the hiding or collapsing of sections of source code. The elision is rule-based which allows the user to customize the views by combining elision strategies. Elision is a technique that has been used by mainframe editors for many years.

Syntax-directed editing is another technique that might assist programmers in software development and maintenance tasks. Syntax- or language-directed editors "combine the text manipulation capabilities of a general-purpose editor with the syntax-checking functions of a compiler" [MORIS81, p. 28]. The main advantage of language-directed editors is that they facilitate more productive editing sessions by helping to reduce syntax errors. This, in turn, helps to make programs free of compile time errors. One of the main drawbacks to syntax-directed editing is that it is difficult to relax the syntax rules to permit unrestricted editing. Raymond claims that the other disadvantages of syntax-directed editing are 1) the structured constraints typically cause more problems than they solve, 2)

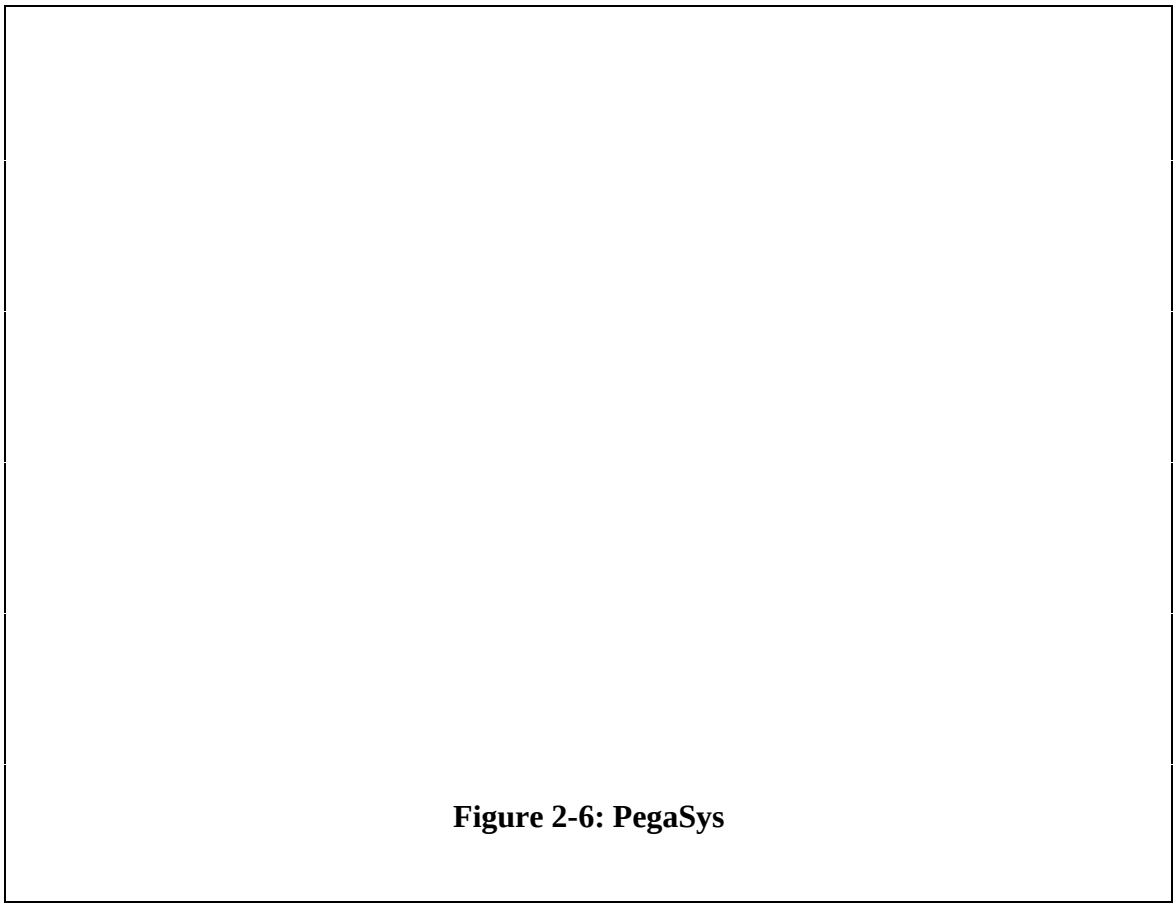

Figure 2-6: PegaSys

navigating the structure hierarchy is often harder than navigating straight text, and 3) formally specifying the text structure is a cumbersome and error prone operation [RAYMO88].

An interesting alternative to syntax-directed editing is literate programming which advocates writing programs the way they are explained. Literate programming promotes the interleaving of documentation or commentary with code so that the code is rearranged to better appeal to readers. Gurari and Wu discuss a WYSIWYG literate programming

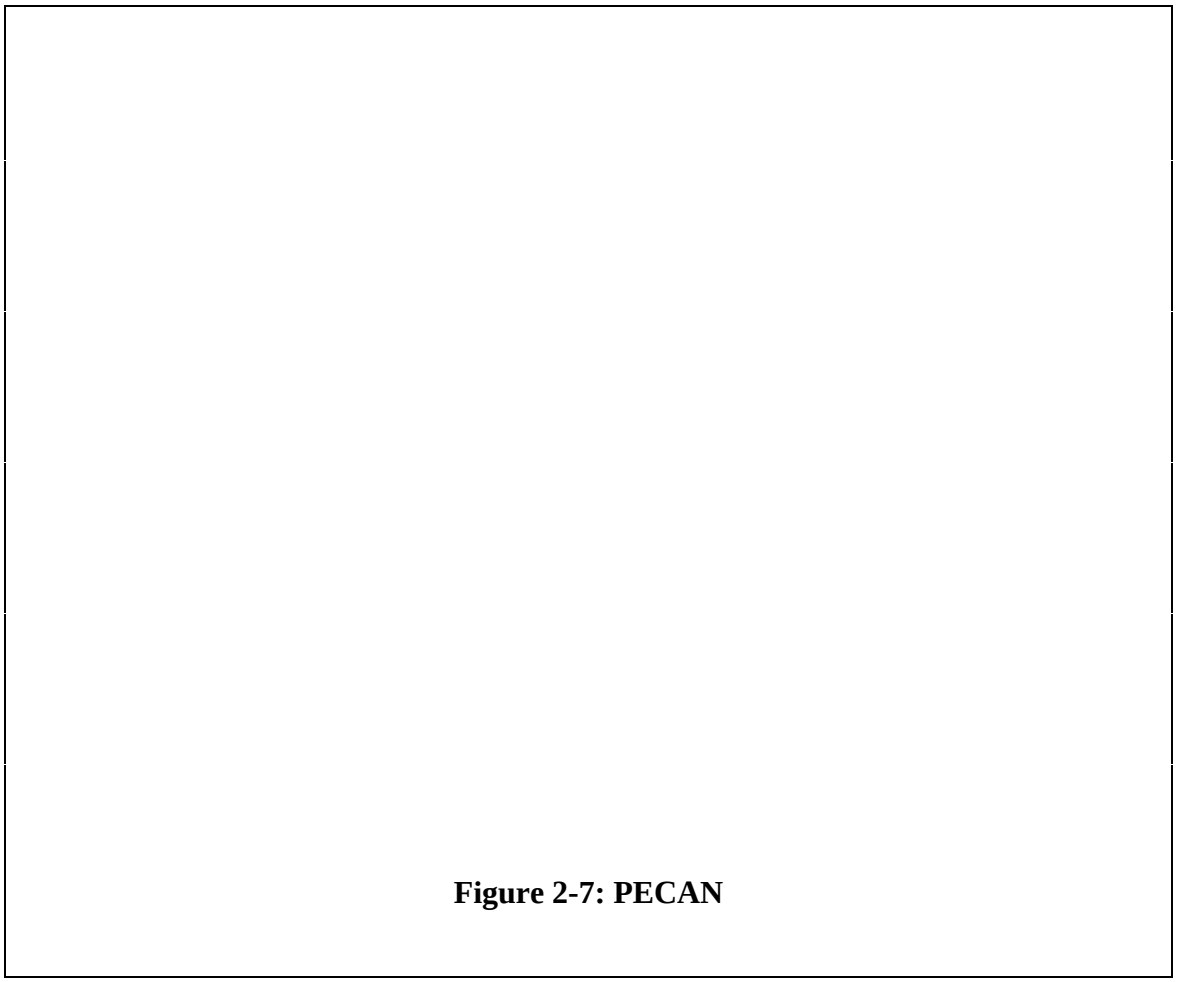

Figure 2-7: PECAN

system that distinguishes between code and prose by the fonts that they are written in [GURAR91]. This system can extract and save the code for later compilation.

Despite the numerous techniques and tools that have been developed in attempts to improve programmer productivity and to make source code easier to understand and maintain, many difficulties still exist. The goal of this research was to produce a prototype system that addressed the issues of multiple editable source code views, extensibility, and multiple external tools as information sources.

2.2 Research Justification

The high costs of software maintenance and development has motivated the development of a large number of applications such as CASE tools. Many of these tools have been designed to improve the productivity of software developers and maintainers by assisting in program comprehension. These tools range from simple, single view source code browsers, to multi-view systems that present information from a variety of sources.

Much of the empirical work to date has focused on the benefits of basic, source code browsing tools that simply provide a mechanism for examining the code in a non-editable manner. More work is required to develop tools that provide users with editable views of the source code. Such tools allow software developers and maintainers to view auxiliary information and immediately react to it without switching to a separate application to edit the source code. Another important issue that requires further examination, is providing multiple source code views. Without multiple views, a variety of tools must be used to display information that could have been presented by a single tool using multiple views.

Virtually all of the tools that were examined in preparation for this work presented intrinsic program information to the user. A few tools were encountered that presented extrinsic information, but most of these created their program views using data from fixed sources. For example, a tool might be able to display data from a specific version control system, but not from other tools. A customizable system with the ability to create source code views using data from virtually any external tool is more useful. By using custom

data sources to create the program views, the learning curve for this type of tool would be reduced since the developer's existing tools can be used as information sources. As well, the user can determine the information that is important and create views based on it. With existing tools, the user might need to combine a number of tools to achieve a result that may not be exactly what is desired.

In addition to defining the information to be contained in the source code view, it is also desirable to define how this information is presented to the user. It would be impossible to support all of the ways that information could be conveyed to the user, however, some flexibility should be provided by applications to allow the user some control over how data or information about a program is presented. Few tools currently available provide this type of flexibility.

Finally, software development and maintenance tools must be extensible. The user must be able to customize the application to suit his or her specific development or maintenance needs. The tools that were examined provided little or no customizability.

This research project was undertaken to address these limitations in existing software development and maintenance tools. The goal of this work was to design a prototype system that would assist software developers and maintainers in understanding a program's source code by presenting arbitrary sets of extrinsic information about the program in editable source code views.

2.3 System Overview

A collection of tools was designed and implemented that together provide software developers and maintainers with editable views of source code and auxiliary information. The system uses ancillary data from external tools to markup a source code file with tags corresponding to this information (Sections 3.2, 3.3, 3.4, and 3.5.1). These tags are replaced with customized, editor-specific tags that apply colour and font changes as well as text elision to the source code (Sections 3.5.2 and 3.6). The user can load this enhanced source code file into the system's editor to modify the program while viewing the embellishments that correspond to the auxiliary information (Section 3.7). The user can customize the source code views to display relevant and important information in the most effective manner.

Chapter 3

Implementation

In this chapter we discuss the implementation of the system. We begin with a high-level overview of the organization followed by a more in-depth discussion of each of the main components. Interested readers should refer to Appendices A, B, and C for specific details regarding the format of the auxiliary data file, the tag map file, and the intermediate tagging language respectively.

3.1 Overview

The system was developed as a collection of tools that work together to provide modifiable visualizations of source code. For extensibility, we built a set of tools rather than a single application to handle different source code languages, tagging schemes, and embellishment types. The modular design of the system makes this possible.

This system consists of the source code and auxiliary data file parsers, the tag generator, the tag replacement and removal tools, and the source code editor. Figure 3-1 shows the architecture of the editable views system.

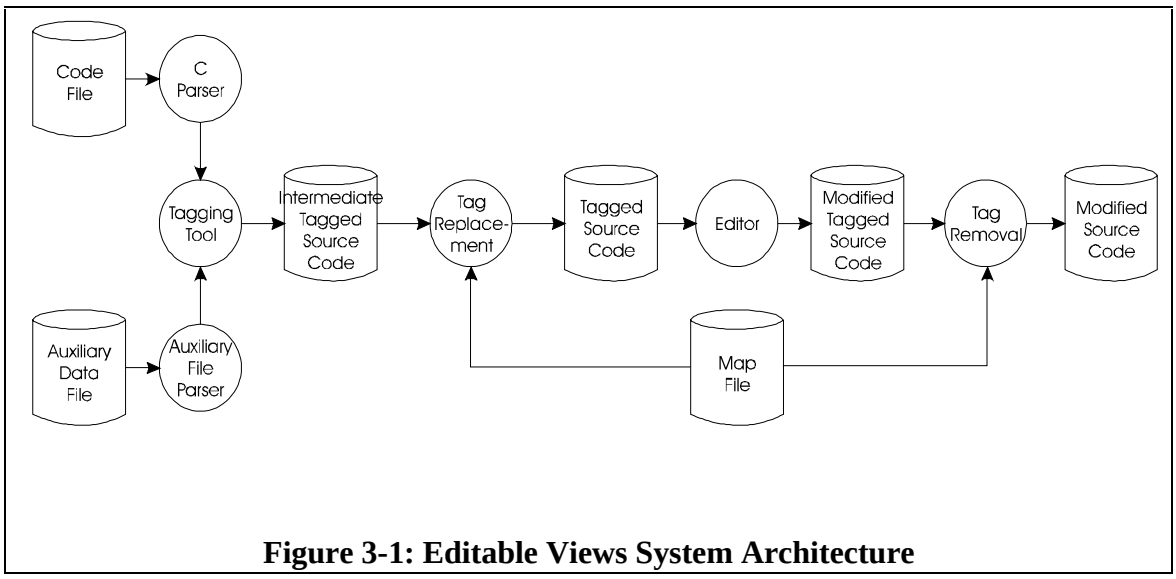

Figure 3-1: Editable Views System Architecture

The code file contains the source code that the user wishes to embellish and edit. The auxiliary data file can contain information from one or more of a variety of existing external tools such as profilers, version control systems, and test-coverage tools. The auxiliary data file defines how information is mapped to the source code via the tagging mechanism. This file is discussed in greater detail in Section 3.2.

Two Lex/YACC parsers, the C parser and the auxiliary data file parser, read in the source code and auxiliary data. The tagging tool uses the information contained in the auxiliary data file to tag the source code file with external and syntactic information. The

source code parser is discussed in Section 3.3. The tagging tool is discussed in detail in Section 3.4 while Section 3.5 describes the tagging languages that are used in the system.

Once the source code file has been tagged the tag replacement tool converts the system's tags to tags appropriate to the specific editor being used. This tool uses a Lex/YACC parser to read a map file specifying how this mapping should be done. This layer of indirection provides editor independence. The tag replacement tool and its associated map file are discussed in Section 3.6.

Editing of the source can be performed after it has been tagged using the editor's tagging language. The editor currently provided by the system is FrameMaker®. This is a document preparation application that provides a wide variety of text embellishment facilities as well as a full-featured tagging language. The editor embellishes the text of the source code using changes in fonts and colour as well as text elision. The embellishments are based on the user's specification, contained in the map file, of how the various tagged sections of source code are to be handled. A more detailed discussion of the editor and its tagging language is contained in Section 3.7.

Once the user has completed the editing of the tagged source code and saved the modified file, the tags can be removed using the tag removal tool. This leaves the modified source code ready to be compiled, profiled, etc. Similar to the tag replacement tool, the tag removal tool uses a Lex/YACC parser to read the map file that specifies how

this mapping should be done. The tag removal tool and its map file are discussed in Section 3.6.

The editable views system provides a convenient general-purpose facility for visually mapping information from external tools to source code. The result is an embellished, editable view of the code that assists in understanding source code. By allowing the use of existing tools to create the source code views, our system is non-intrusive on the normal compile-run-edit-test cycle of software development.

The following chapters in this section expand on the previous description of the system. The design and implementation of the different aspects of the system are described in greater detail. Problems that were encountered are discussed and their solutions, if found, are described. This chapter concludes by describing problems that still exist in the system and suggestions for how to overcome these problems in possible future versions of the software.

3.2 The Auxiliary Data File

The auxiliary data file has been designed to encapsulate information from a broad variety of external software development or maintenance tools. Examples of such tools include profilers, compilers, and version control systems. Many of these tools produce information that is line-oriented. In other words, the information may attach some data to any line of source code in a program. In version control systems, for example, information

regarding when a source file has been modified and who made the changes, are determined for each line of source code. The auxiliary data file can be used to attach line-oriented data from a number of tools to the source code. Data from the external tools must be converted into the auxiliary data file format by the user. This should be a relatively simple process to automate. For a complete specification of the auxiliary data file see Appendix A.

The auxiliary data file consists of four sections: the data section, the value mapping section, the label mapping section, and the priorities section. The data section indicates how values and labels are associated with lines and identifiers. Figure 3-2 shows an example of the data section of an auxiliary data file. A line of the form

```
1 - 15          12.6
```

associates the value 12.6 with the lines one through fifteen. The following line

```
nTestInt        VARIABLE
```

associates the label `VARIABLE` with the identifier `nTestInt`.

```
data:

     1 - 15           12.6
     nTestInt         VARIABLE
```

Figure 3-2: Auxiliary Data File - Data Section

The value mapping section defines how values correspond to the tags that are used to markup the source code. Figure 3-3 shows an example of the value mapping section of an auxiliary data file. This example indicates that values in the range 43.6 through 50.9 are

assigned a start tag with text RANGE_START and an end tag with text RANGE_END. Therefore, any lines or identifiers that have a value in this range associated with them, will be tagged using these start and end tags.

```
values:
      43.6 - 50.9       RANGE_START       RANGE_END
```

Figure 3-3: Auxiliary Data File - Value Mapping Section

The label mapping section is similar to the value mapping section except that it defines how labels correspond to tags used to markup the source code. Figure 3-4 shows an example of the label mapping section from an auxiliary data file. This example indicates that the label MYLABEL is assigned a start tag with the text LABEL_START and an end tag with the text LABEL_END. Therefore, any lines or identifiers that have the label MYLABEL associated with them, will be tagged using these start and end tags.

```
labels:
      MYLABEL      LABEL_START       LABEL_END
```

Figure 3-4: Auxiliary Data File - Label Mapping Section

The priorities section of the auxiliary data file defines the priorities of different syntactic constructs. This determines how the tagging module will tag the constructs when they are encountered in the source code file; constructs with higher priorities getting precedence over constructs with lower priorities. For example, if a construct with priority four occurs within a construct of priority five, the priority four construct will not be

tagged but the priority five construct will. Figure 3-5 shows an example of the priorities section of an auxiliary data file. Constructs with negative priorities (-1 for example) are never tagged. The priorities defined in this section will only have an effect on tagging if prioritized tagging is enabled. Prioritized tagging is discussed in detail in Section 3.4 on The Tagging Tool.

```
priorities:
      while_headers     2
      while_bodies      -1
```

Figure 3-5: Auxiliary Data File - Priorities Section

There are three forms of tagging supported by the system; standard, stack-based, and prioritized. Standard tagging places a start-end tag pair around every taggable construct in the source code file. Stack-based tagging maintains a stack of active constructs (i.e. constructs that have been tagged with a start tag, but not yet completed with an end tag). Prioritized tagging uses a tag state stack as in stack-based tagging, in conjunction with priorities to restrict tag placement as mentioned earlier. The tagging strategies are discussed in more detail in Section 3.4 on The Tagging Tool.

The auxiliary data file is designed to contain information about a program from external tools such as profilers, compilers, or version control systems. The user is responsible for converting this information into the auxiliary data file which is then used to tag source code files. Figure 3-6 shows an example of a complete auxiliary data file. This example shows the type of information that could be created from profiler data. In this

case, the profiler has associated the value 0.0 with lines 1 through 5, 10 through 12, and 15 through 17. Lines 6 through 9, have the value 0.25 associated with them and so on. The user has decided that lines with an associated value of 0.0 will be surrounded by the tags NONE_START and NONE_END while lines with values between 0.1 and 0.49 will be surrounded by the tags LOW_START and LOW_END. Similarly, lines with associated values between 0.5 and 0.99 will be surrounded by the tags MEDIUM_START and MEDIUM_END. Finally, HIGH_START and HIGH_END tags are to be used to surround lines with associated values between 1.0 and 2.0. No labels are being used by the profiler so this section is empty. The user has decided that line tagging will have priority 5, the bodies of *if* statements priority 4, and the headers of *if* statements will not be tagged.

```
data:
      1 - 5             0.0
      6 - 9             0.25
      10 - 12           0.0
      13 - 14           1.65
      15 - 17           0.0
      18                1.76
      19 - 20           0.56
      21                1.3
      22                0.27

values:
      0.0               NONE_START        NONE_END
      0.1 - 0.49        LOW_START         LOW_END
      0.5 - 0.99        MEDIUM_START      MEDIUM_END
      1.0 - 2.0         HIGH_START        HIGH_END

labels:

priorities:
      lines             5
      if_bodies         4
      if_headers        -1
```

Figure 3-6: Auxiliary Data File - Complete File

3.3 The C Parser

The C parser is responsible for reading a syntactically valid C source code file. As the parser reads the file, it locates the start and end of constructs[2] within the source code. With this information, the tagging module places start and end tags around sections of the source code. The C constructs that are currently identified by the parser include

[2]The term construct refers to an entity or building block of the language. Examples for C include *for* loops and *if* statements.

comments, variables (including function names), variable declarations, type definitions, function headers, function calls, *if* statements (both headers and bodies), loop (*for, do*, and *while*) statements (both headers and bodies), *switch* statements (both headers and bodies), and jump (*break, continue, goto,* and *return*) statements.

The source code parser used in the project is a modified version of a public domain ANSI C parser available via anonymous ftp from ftp.uu.net. The grammar and scanner are contained in /pub/usenet/net.sources/ansi.c.grammar.Z at this site. The parser as provided was insufficient in certain key areas.

Some of the following issues needed to be addressed before the parser was a useable tool. Initially, the parser did not recognize C pre-processor constructs and ignored comments. The parser was easily modified to recognize, but ignore, pre-processor constructs and to identify the start and end of comments. Certain pre-processor constructs that needed to be handled were given to a special C pre-processor parser that is described later.

The original parser was also lacking a type table. The parser, when it encountered an identifier, was unable to determine whether the identifier represented a variable or a user-defined type. It was therefore necessary to build a type table prior to parsing the source code file. The naive approach to solving this problem is to build a type table on the fly by adding to the table as *typedef*'s are encountered in the code file. Clearly this will not work since type definitions are frequently contained in external files (e.g. .h files). This suggests

another solution. Each included file can be opened and the type definitions extracted prior to parsing the C code. A way to simplify the collection of type information in this approach is to use the C pre-processor itself.

The C pre-processor takes a C source code file as input and creates a fully expanded version of the same file that has all *#include* files included, all macros (i.e. *#define*'s) expanded, and all *#ifdef*'d code included or removed. In addition, for each *#include*'d file there is a line in the fully expanded file of the form

```
# xxx "...."
```

that contains the full path specification of the included file. *xxx* is a number and the portion in quotes is the path of the included file. The C pre-processor can be used to create a temporary file that can be parsed to collect a list of the types defined in the main source code file and all *#include* files. There are two issues to be addressed with this approach. The first is that the type table is built prior to parsing the original source code file. This poses no problem, however, since types cannot be redefined provided that the source code is syntactically correct (this restriction is explained later). The second issue is that types may be defined using macros. An example of this is

```
#define SINT short int
```

It is then possible to define a variable using this macro,

```
SINT test;
```

for example. This poses a problem, since the C pre-processor expands all macros. Types defined in this way cannot be identified using the C pre-processor. The only way to locate these types is to scan the source code file and each included file for any lines of the form

```
#define identifier type
```

The type table for the application is built by first creating a temporary file using the C pre-processor. This file is parsed using a simple Lex/YACC parser that extracts type definitions and adds them to the type table. Each *#include* file is then opened and searched for type definitions that occur within macros. The macro names are added to the type table as they are encountered. There are two minor problems with this approach. The first problem is that while a macro definition of the form

```
#define A B
#define B int
```

is legal in C, the type table we produce will only contain B as a type and not A since the macro A is encountered before the macro B. The C pre-processor resolves macro definitions after the file has been completed parsed. The second problem occurs if the user specifies a type within a macro in a strange way, for example

```
#define A /* my macro */ int
```

In such a case, the name will not be added to the type table. The reason for both of these problems is simply that macros can be extremely complicated and the effort required to handle complicated macro parsing outweigh the benefits in this case. Rather than develop a complicated macro parser, a pragmatic approach was taken that requires users to write code with a reasonable amount of proper style. Complicated macros, especially those

containing type definitions, are not typically considered part of a good programming style. The best way to ensure that type definitions are located during our system's parsing is to use *typedef*'s, rather than macros, to define them.

Once the type table has been built, the C parser is responsible for identifying the beginning and end of various syntactic constructs within the source code. When the beginning of a taggable construct is encountered, a call-back routine in the tagging module is invoked that indicates to the tagging module which construct has been encountered and the token(s) that mark the beginning of it. Similarly, when a complete construct is encountered, the parser invokes another call-back routine to indicate that the construct has been completed. This approach of using call-back routines allows the tagging module to place start and end tags correctly around constructs in the source code file. The source code scanner adds text to a buffer as the source code is scanned. When the call-back function is invoked, the execution of the parser is suspended and the tagging module can move through the buffer to place start or end tag(s). Once the call-back routine returns to the parser, the start or end tag for the current construct has been correctly placed, and the execution of the parser resumes. For example, suppose the source code to be parsed is

```
if[1] (x == 1)[2]
   x++;[3]
```

Several things occur as the parser reaches the following locations in the code (the superscript numbers indicate the location of the parser):

1. - the keyword *if* is detected, marking the beginning of the constructs **if_statement** and **if_header**

- call-back routines are invoked indicating that the constructs **if_statement** and **if_header** have been detected
- tags are placed (if required) to mark the start of the **if_statement** and **if_header** constructs

2. - the right parenthesis token is detected which marks the end of the construct **if_header** and the beginning of the construct **if_body**
   - call-back routines are invoked to indicate that the construct **if_header** has ended and the construct **if_body** has started
   - tags are placed (if required) to mark the end of the **if_header** and the start of the **if_body** constructs
3. - the semicolon token is detected which marks the end of the constructs **if_body** and **if_statement**
   - call-back routines are invoked to indicate that the constructs **if_body** and **if_statement** have ended
   - tags are placed (if required) to mark the end of the **if_body** and **if_statement** constructs

Once this procedure is complete, the resulting tagged text might appear as follows

```
%[IF_START%]%[IH_START%]if (x == 1)%[IH_END%]%[IB_START%]
   x++;%[IB_END%]%[IF_END%]
```

In order to make this approach work properly, it was necessary to restructure certain rules in the grammar of the parser to avoid conflicts that would occur when code was inserted to invoke the call-back routines. For example, the rule for a function definition in the original C grammar was:

```
func_def : decltr cmpd_stmt
         | decltr dec_list cmpd_stmt
         | dec_specs decltr cmpd_stmt
         | dec_specs decltr dec_list cmpd_stmt;
```

In order to tag function headers properly, this was changed to:

```
func_def : func_head func_body;

func_head: decltr {inserted code}
         | dec_specs decltr {inserted code};

func_body: cmpd_stmt
         | dec_list cmpd_stmt;
```

The original rule with the inserted code to perform the tagging would have been

```
func_def : decltr {inserted code} cmpd_stmt
         | decltr {inserted code} dec_list cmpd_stmt
         | dec_specs decltr {inserted code} cmpd_stmt
         | dec_specs decltr {inserted code} dec_list cmpd_stmt;
```

which is ambiguous with respect to the inserted code. That is, when the parser must execute the inserted code, it does not know which rule reduction it is in. When the parser sees a `decltr`, it may be in the `decltr cmpd_stmt` reduction or the `decltr dec_list cmpd_stmt` reduction. Which reduction is active will not be known until the parser sees a `cmpd_stmt` or a `dec_list`.

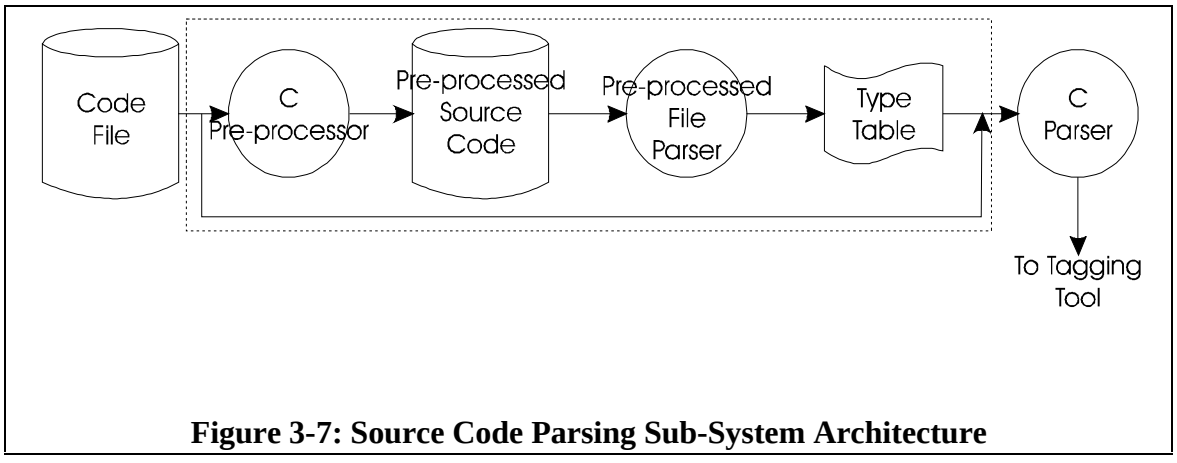

Figure 3-7: Source Code Parsing Sub-System Architecture

Figure 3-7 shows how the source code parsing components of the system are organized. See Figure 3-1 for an overview of the entire system architecture.

We summarize here the outstanding issues with the parser that have been mentioned. The first is that in order to guarantee that types are correctly located, *typedef*'s or only very simple macros should be used. The second consideration is that since the parser in the system works on the non-pre-processed source code, code that uses complicated and unconventional macros may not be tagged correctly, and conditional code will also be subject to tagging. Again, this should not be a great concern provided that the programmer does not make excessive use of complicated, pre-processor constructs. The fact that the parser deals with non-pre-processed source code is perfectly reasonable since we will eventually be editing a file in its native form. Clearly, the editor must not deal with an expanded version of the file. Finally, the source code must be syntactically correct for it to be correctly parsed and tagged. This is because the parser, as provided, terminates when it encounters a syntax error. This is actually a positive feature of the system since it detects syntax errors prior to compilation. In addition, the primary benefit of this system is the ability to visualize data from external tools, most of which also require syntactically correct, possibly compilable code.

In conclusion, the ANSI C parser within the editable views system uses the C pre-processor to create a temporary, fully expanded version of the source code file. This file is parsed to build an internal type table. All included files are then scanned to locate types

defined within macros. These types are also added to the type table. The original, unexpanded source code file is then parsed and the beginning and end of syntactic constructs are located. This information is provided to the tagging module through the use of call-back routines.

3.4 The Tagging Tool

The tagging tool handles the correct placement of start and end tags around sections of the source code. To accomplish this task, the tagging module communicates with the C parser through the use of call-back functions, which are used by the parser to indicate the start and end of various syntactic constructs within the source code.

The tagging module maintains a dynamic text buffer that accumulates text from the source code file along with tags. This buffer is periodically flushed to the output stream. The buffer must be maintained with an appropriate amount of data in order for the tags to be placed correctly. The reason for this is that to remove or insert tags, it is frequently necessary to backtrack over text that has already been read and processed. The issue of when to empty the buffer is important, because we do not want to clear the buffer before all required tags have been placed in it. However, maintaining an ever-increasing buffer is undesirable, so some heuristic strategy needed to be found. We decided to write the buffer to the output stream after a complete function has been read since at this point, all of the constructs we recognize will terminate, and no more tags will be added to the text

in the buffer. The C parser notifies the tagging module when a complete function has been read through the use of a call-back routine.

The buffer maintained by the tagging module is dynamic and increases in size as the C parser reads text from the source code file and adds it to the buffer. Whenever it is necessary to modify the contents of a buffer, the required increase in size of the buffer is computed and a re-size routine is called. This routine compares the current size of the buffer to the requested size and allocates more space if necessary. When the text in the buffer is written to the output stream, the space for the buffer is deallocated. The size of the buffer is never decreased except when the buffer is written and deallocated. This is not a concern because for the most part, the text in the buffer grows monotonically. Occasionally the quantity of text in the buffer decreases slightly, but since the decrease is small, the overhead required to update the size of the buffer would outweigh any benefits gained from frequently decreasing the size.

A number of general purpose text buffer manipulation routines were implemented to assist with the large amount of text handling required. Routines were written to match pairs of tokens, move to a specific token, move over identifiers or arbitrary text strings, and to locate arbitrary text strings in the buffer. Recursive routines were written to skip

```
#include <stdio.h>

/*
 * Simple function.
 */
void main() {

   int x;

   /*
    * Simple loop.
    */
   for (x = 0; x < 3; x++) {

      /*
       * Simple if.
       */
      if (x == 2) {
         fprintf(stderr, "X = 2\n");
      }
   }
}
```

Figure 3-8: Sample Source Code - Before Tagging

whitespace[3] in the buffer. These routines are used in special situations that are described later. All of the manipulation routines take, as parameters, a text buffer and a start index into this buffer as input. The routines begin processing at the specified starting location in the buffer. When they finish processing, they return an index into the buffer that corresponds to their terminating location. The use of indices as input and output to the routines allows the routines to invoke each other, enables them to remain general, and simplifies the implementation.

---

[3]Whitespace in this thesis refers to spaces, tab characters, vertical tab characters, new line characters, C comments, and pre-processor constructs.

The tagging module communicates directly with the C parser through the use of two call-back functions. These routines are invoked by the parser when it locates the start or end of a syntactic construct. When the call-back function is entered, the tagging module is responsible for correctly placing the start or end tag for the construct within the text. For more information on the tagging language, see Section 3.5. The procedure for adding tags varies slightly depending on the type of tagging that is in effect and the construct that is being tagged. Recall that there are three forms of tagging that can occur; standard, stack-based, and prioritized.

Standard tagging places a start and end tag pair around every taggable construct in the source code file. Constructs can nest (i.e. contain other constructs) and they are tagged in a nested manner; that is, the tags for the constructs do not overlap. Figure 3-8 shows some sample source code prior to tagging. Figure 3-9 shows the same source code after standard tagging has been performed on it. Standard tagging is sufficient if the editor being used to view the tagged document is capable of saving and restoring states when changing embellishments. For example, suppose the editor provides a `START_ITALIC` tag and an `END_ITALIC` tag for marking italic text. When it encounters the `START_ITALIC` tag the editor saves the current state and turns italics on. When the `END_ITALIC` tag is encountered, the editor restores the formatting that was active before the `START_ITALIC` tag was encountered.

For editors that do not provide this form of state saving, the tagging can be performed using a state stack. With this form of tagging, the tagging module maintains a stack of constructs that are in the process of being tagged. When a new construct is encountered, the construct is pushed onto the stack and its start tag is inserted into the text. When the end of a taggable construct is located, the end tag for the construct is inserted into the text and the construct is popped from the top of the stack. If the stack is not empty at this point, a start tag for the construct on the top of the stack is then inserted into the text. Figure 3-10 shows the sample code after undergoing stack-based tagging. Note that when

```
#include <stdio.h>

%[COMMENT_START%]/*
 * Simple function.
 */%[COMMENT_END%]
%[FHEADER_START%]void main()%[FHEADER_END%] %[CMPND_START%]{%[CB_START%]

   %[DECL_START%]int x;%[DECL_END%]

   %[COMMENT_START%]/*
    * Simple loop.
    */%[COMMENT_END%]
   %[FOR_START%]%[FH_START%]for (x = 0; x < 3;
x++)%[FH_END%]%[FB_START%] %[CMPND_START%]{%[CB_START%]

      %[COMMENT_START%]/*
       * Simple if.
       */%[COMMENT_END%]
      %[IF_START%]%[IH_START%]if (x == 2)%[IH_END%]%[IB_START%]
%[CMPND_START%]{%[CB_START%]
         %[FCALL_START%]fprintf(stderr, %[STRING_START%]"X =
2\n"%[STRING_END%])%[FCALL_END%];
      %[CB_END%]}%[CMPND_END%]%[IB_END%]%[IF_END%]
   %[CB_END%]}%[CMPND_END%]%[FB_END%]%[FOR_END%]
%[CB_END%]}%[CMPND_END%]
```

Figure 3-9: Sample Source Code - After Standard Tagging

an end tag for a construct is placed, there is a start tag for the active construct immediately following it. This method of tagging is good for editors that cannot save state but simply turn different types of formatting on and off. For example, when the editor encounters an `START_ITALIC` tag it turns italics on. When the `END_ITALIC` tag is seen, the editor simply turns italics off. Notice that there are more start tags than end tags using this type of tagging. This is not a problem, however, since the end tags are simply used to return to a default mode of formatting. If this method of tagging is being used, the start tags are responsible for setting the embellishment type. As a result, even if matching end tags were inserted for each start tag, they would have no effect.

The final method of tagging is to use priorities in conjunction with the state stack. In this mode, each construct is assigned a priority and it is this priority that determines whether a construct will be tagged. If a construct is not explicitly given a priority, then a common, default priority is assigned. The stack also has a priority that is defined to be the highest priority of the constructs currently on the stack. When the stack is empty, the priority of the stack is set to the default. Essentially, lower priority constructs will not be tagged if they are contained within higher priority constructs. The tagging in this mode operates similarly to that in the stack-based tagging with two exceptions. First, when the start of a construct is encountered, the priority of the construct is compared to the priority of the stack. If the construct priority is greater than or equal to the stack priority, the construct is pushed on the stack, the start tag is inserted into the text, and the stack

```
#include <stdio.h>

%[COMMENT_START%]/*
 * Simple function.
 */%[COMMENT_END%]
%[FHEADER_START%]void main()%[FHEADER_END%] %[CMPND_START%]{%[CB_START%]

   %[DECL_START%]int x;%[DECL_END%]%[CB_START%]

   %[COMMENT_START%]/*
    * Simple loop.
    */%[COMMENT_END%]%[CB_START%]
   %[FOR_START%]%[FH_START%]for (x = 0; x < 3;
x++)%[FH_END%]%[FOR_START%]%[FB_START%] %[CMPND_START%]{%[CB_START%]

      %[COMMENT_START%]/*
       * Simple if.
       */%[COMMENT_END%]%[CB_START%]
      %[IF_START%]%[IH_START%]if (x ==
2)%[IH_END%]%[IF_START%]%[IB_START%] %[CMPND_START%]{%[CB_START%]
         %[FCALL_START%]fprintf(stderr, %[STRING_START%]"X =
2\n"%[STRING_END%]%[FCALL_START%])%[FCALL_END%]%[CB_START%];

%[CB_END%]%[CMPND_START%]}%[IF_END%]%[CB_START%]%[CMPND_END%]%[CB_START%
]%[IB_END%]%[CB_START%]

%[CB_END%]%[CMPND_START%]}%[CMPND_END%]%[FB_START%]%[FB_END%]%[FOR_START
%]%[FOR_END%]%[CB_START%]
%[CB_END%]%[CMPND_START%]}%[CMPND_END%]
```

Figure 3-10: Sample Source Code - After Stack-Based Tagging

priority is updated. If the priority of the construct is less than the priority of the stack then the construct is not tagged at this time. Therefore, the user can prevent constructs from being tagged by assigning them priorities that are less than the default of one. The second difference is that when the end of a construct is located, if a start tag was placed for the construct, then the corresponding end tag for the construct is inserted into the text, the construct is popped off the stack and the priority of the stack is updated. If the stack is not empty, a start tag for the construct on the top of the stack is inserted into the text.

Figure 3-11 shows the sample code tagged with priorities set at -1 for compound statements and bodies, loops, and *if* statements and 5 for loop bodies and headers, and 1 for the default. Notice how the comment block and the *if* statement within the loop have not been tagged because their priorities are lower than the priority of the loop body. This tagging scheme provides all of the benefits of stack-based tagging, and allows the user to highlight constructs based on their importance. Priorities are assigned to constructs through entries in the auxiliary data file. For more information on the auxiliary data file, see Section 3.2.

The different tagging schemes provided by the system allow the system to be flexible by remaining editor-independent. The tagging scheme can be chosen that best matches the capabilities of the editor used.

The procedure for actually inserting a tag into the text is straight-forward for most constructs. First, the position at which to insert the tag is located. This may involve moving backwards through the text buffer. The tag for the construct is then obtained through lookup in a table. The buffer is re-sized to accommodate the new tag and then split in two at the insertion point. The tag is appended to the first portion of the buffer and then the second portion is appended to this. Some notable exceptions to this procedure occur when tagging function calls, *if* statements, and function headers.

```
#include <stdio.h>

%[COMMENT_START%]/*
 * Simple function.
 */%[COMMENT_END%]
%[FHEADER_START%]void main()%[FHEADER_END%] {

   %[DECL_START%]int x;%[DECL_END%]

   %[COMMENT_START%]/*
    * Simple loop.
    */%[COMMENT_END%]
   %[FH_START%]for (x = 0; x < 3; x++)%[FH_END%]%[FB_START%] {

      /*
       * Simple if.
       */
      if (x == 2) {
         fprintf(stderr, "X = 2\n");
      }
   }%[FB_END%]
}
```

Figure 3-11: Sample Source Code - After Prioritized Tagging

Function call tagging is complicated by the fact that the parser does not recognize the function call until the parameter list for the function call has been encountered. Therefore, to correctly place the start tag for the function call, the start of the function call must be located by backtracking through the buffer. A recursive function was written to locate the beginning of a function call. This function identifies expressions that precede the parameter list. The expressions that are identified can be of the following forms

1. identifier (e.g. `fcall(args)`)
2. (expression) (e.g. `(fcall_ptr)(args)`)
3. $expression_1$[$expression_2$] (e.g. `fcall_array[index](args)`)
4. expression.identifier (e.g. `struct.fcall(args)`)
5. expression->identifier (e.g. `struct_ptr->fcall(args)`)
6. expression++ (e.g. `fcall_ptr++(args)`)
7. expression-- (e.g. `fcall_ptr--(args)`)

If the expression is of the first type, *identifier*, then the start of the function call is simply the start of *identifier*. The second form, *(expression)*, is similar. The start of this form of function call is found by matching parentheses around *expression*. The start of the function call is the start of this parenthesized expression. The third form, $expression_1[expression_2]$, requires that the bracketed $expression_2$ be skipped and the start of the function call be located by recursively invoking the function on $expression_1$. If the parameter list is prefixed by *expression.identifier* or *expression->identifer* (forms four and five), then the start of the function call is located by moving backwards past the . or -> and recursively invoking the function on *expression*. Finally, if the expression is of the form *expression*++ or *expression*--, the start of the function call is located by moving backwards past the ++ or -- and recursively invoking the function on *expression*.

With *if* statements, difficulties arise because the ANSI C grammar contains an ambiguity in the way it defines *if* statements. An *if* statement is defined as follows:

```
if_statement   : if (expression) statement
               | if (expression) statement else statement
```

Therefore, the parser does not know which form of *if* statement it has located until it does or does not see the *else* keyword. Because of this ambiguity, the scanner may be well beyond the end of the *if* statement before the parser has recognized the end of the construct. As a result, before placing the end tag for the *if* construct, the tagging module must determine whether it is necessary to move backwards in the buffer over *whitespace* before placing the end tag. If the construct just completed is an *if* statement rather than an

if-else, a special-purpose routine is called to move over *whitespace* to the end of the *if* construct where the end tag is then placed.

Finally, function headers require special treatment while being tagged, but only if prioritized tagging is being used. With function headers, the parameter list is tagged (as a declaration) before the function header construct is recognized by the parser. As a result, if the priority of function headers is greater than that of variable declarations, the tags surrounding the parameter list must be removed. A routine was written to remove erroneously placed tags from a function header. This routine is called when the end tag for a function header is placed by the tagging module.

In summary, the tagging module is responsible for correctly placing start and end tags around constructs in the C source code file. The tagging tool receives information about the location of the constructs from the C parser via call-back functions. The tagging module supports three forms of tagging; standard, stack-based, and prioritized, of which one can be selected by the user. Standard tagging provides no state saving mechanism while stack-based tagging does. Prioritized tagging allows the user to assign priorities to give precedence to certain constructs. The priorities of the constructs are set in the auxiliary data file.

3.5 The Tagging Languages

There are two distinct tagging languages that are used in the system, the Intermediate Tagging Language (ITL) and the editor tagging language which is currently the Maker Interchange Format (MIF). The Intermediate Tagging Language is used internally by the system to tag constructs within the C source code file. Section 3.5.1 describes this tagging language. The Maker Interchange Format is used by the editor of the system (FrameMaker). MIF is discussed in Section 3.5.2. The second tagging language depends on the editor being used with the system. All that is required is that the editor have an accessible, ASCII-tagged storage format. After the source code has been tagged using ITL, the tags are replaced with MIF tags using the tag replacement tool so that the editor can be used to view the tagged document. See Section 3.6 for more information on the tag replacement tool.

The decision to use a custom, non-proprietary tagging language to markup the code and then replace these tags prior to editing was made for two reasons. First, by using a custom tagging language, the implementation of the tagging component of the system was simplified. The flexibility of a complicated system such as SGML was not required for the task of marking up the source code. In addition, SGML tools were not very sophisticated at the time this research began. The Intermediate Tagging Language was simple to implement and is adequate for the markup task. The second reason a non-proprietary rather than commercial system was chosen is that by providing a mechanism for replacing

the intermediate tags, the system remains flexible. Different views of the source code can be created by changing how the tags are replaced. Also, different editors can be used to view the source code simply by creating different tag replacement map files for each editor.

3.5.1 Intermediate Tagging Language (ITL)

The Intermediate Tagging Language (ITL) is the tagging language used by the system to markup the C source code. ITL tags are placed around constructs by the tagging module. For more information on the tagging component of the system, see Section 3.4. Each ITL tag has the form

```
%[<construct>_[START | END]%]
```

ITL tags use the character sequence *%[* to denote the start of a tag and the sequence *%]* to denote the end of the tag. These sequences were chosen simply because they are easy to parse and do not occur in a reasonable selection of programming languages. The text contained between the tag delimiters indicates the construct being tagged and whether the tag represents the start or end tag for the construct. For example, the tag

```
%[FOR_BODY_START%]
```

is used to mark the start of the body of a *for* loop. Similarly, the end tag for the same construct is

```
%[FOR_BODY_END%]
```

A complete list of the constructs and their corresponding start and end tags is given in Appendix C. Figure 3-9 shows sample source code tagged using the Intermediate Tagging Language.

3.5.2 Maker Interchange Format (MIF)

Maker Interchange Format (MIF) is a tagging language provided by FrameMaker. MIF can provide a complete representation of a FrameMaker document and as a result supports all of the text, graphics, formatting, and layout constructs that FrameMaker understands. A MIF file consists of a number of statements that describe the structure, layout, and content of the document. MIF statements are of the form

```
<token data>
```

Where *token* represents one of the MIF keywords and *data* represents one or more numbers, a string, a keyword, or nested markup statements. The MIF specification requires that certain special characters in the text of the document be represented by escaped backslash (\) sequences. The following substitutions are required:

Character:	**Representation:**
Tab	\t
>	\>
’	\q
‘	\Q
\	\\

For example, the text “This is a ‘quote’” would be represented as “This is a \qquote\Q”.

For a complete discussion of the MIF file format see [FRAME92a]. Figure 3-12 shows a sample MIF document and Figure 3-13 shows how this document might look in FrameMaker.

3.6 Tag Replacement and Removal

Once the source code has been processed by the parser and tagging components of the system, the tagged source code file must be prepared for manipulation with the editor. This requires that the intermediate tags be replaced with formatting tags that the editor can understand. The tag replacement tool was written to accomplish this.

The tag replacement tool reads information from the tag map file using a simple Lex/YACC parser. This file describes how ITL tags correspond to the tags used by the editor and how special characters such as tabs, newlines, etc. should be handled. The map file also specifies text that should be inserted at the beginning and end of the final, tagged file. Once the tag replacement information has been extracted from the map file, the tag replacement tool generates a SED script that when executed, replaces the ITL tags with the editor’s tags, replaces special characters as necessary, and inserts the leader and trailer text specified in the map file. The result is a tagged version of the source code file that

```
<MIFFile 3.0> # Identifies this as an MIF file.
# The macros below are used only for the second paragraph of
# text to illustrate how they can ease the process of MIF
# generation.
Define(pr,‘<Para’)
define(ep,‘>’)
define(ln,‘<ParaLine <String’)
define(en,‘>>’)
# First paragraph of text.
<Para
  <PgfTag ‘Body’>
  <ParaLine
    <String ‘MIF (Maker Interchange Format) is a group of ’>
  >
<ParaLine
    <String ‘ statements that describe all text and graphics ’>
  >
<ParaLine
    <String ‘ understood by FrameMaker in an easily parsed, ’>
  >
<ParaLine
    <String ‘ readable text file. MIF provides a way to ’>
  >
<ParaLine
    <String ‘ exchange information between FrameMaker and ’>
  >
<ParaLine
    <String ‘ other applications while preserving graphics, ’>
  >
<ParaLine
    <String ‘ document structure, and format. ’>
  >
>
# Second paragraph of text.
Pr
ln ‘You can write programs that convert graphics or ’ en
ln ‘documents into a MIF file and then import the MIF file ’ en
ln ‘into FrameMaker with the graphics and document format ’ en
ln ‘intact.’ en
ep

# End of MIF file.
```

Figure 3-12: Sample MIF Document

MIF (Maker Interchange Format) is a group of statements that describe all text and graphics understood by FrameMaker in an easily parsed, readable text file. MIF provides a way to exchange information between FrameMaker and other applications while preserving graphics, document structure, and format.

You can write programs that convert graphics or documents into a MIF file and then import the MIF file into FrameMaker with the graphics and document formats intact.

Figure 3-13: FrameMaker Representation Of MIF Document

can be read and viewed using the editor of the system. Figure 3-14 shows a sample map file and Figure 3-15 shows the tagged source code from Figure 3-11 after tag replacement. With the tag replacement tool, it is possible to create map files that provide different views of the source code as well as map files for use with different editors and viewers. For example, if the user wanted a view of the source code that only showed function headers, a map file could be written that only replaces function header tags and removes all others. If the user did not require an editable view, a map file could be created for a document-viewing-only tool such as Mosaic. See Appendix B for a complete description of the format of the map file.

Once the user has finished editing the view of the source code and has saved the file, it may be necessary to remove the tags. If the editor is capable of saving a file without the tags then this will not be necessary. However, if the editor does not provide such a facility, the tag removal tool can be used to accomplish this.

```
leader:
      "<MIFFile 3.00>"
      "include(defaults.mif)"
      "include(formatting.mif)"
      "include(custom.mif)"
      "include(leader.mif)"

trailer:
      "include(trailer.mif)"

pass1:
      start "sp sl ‘"
      end   "’ el ep"
      "\"   "\\"
      "’"   "\q"
      "‘"   "\Q"
      ">"   "\>"
      tab   "\t"

pass2:
      comment_start           "’ el CommentS sl ‘"
      comment_end             "’ el CommentE sl ‘"
      function_header_start   "’ el FuncHS sl ‘"
      function_header_end     "’ el FuncHE sl ‘"
      if_header_start         "’ el IfHS sl ‘"
      if_header_end           "’ el IfHE sl ‘"
      if_body_start           "’ el IfBS sl ‘"
      if_body_end             "’ el IfBE sl ‘"
      "NONE_START"            "’ el SzVTny ColBlk sl ‘"
      "NONE_END"              "’ el SzReg ColBlk sl ‘"
      "LOW_START"             "’ el SzVSml ColCyn sl ‘"
      "LOW_END"               "’ el SzReg ColBlk sl ‘"
      "MEDIUM_START"          "’ el SzMed ColGrn sl ‘"
      "MEDIUM_END"            "’ el SzReg ColBlk sl ‘"
      "HIGH_START"            "’ el SzVBig ColRed sl ‘"
      "HIGH_END"              "’ el SzReg ColBlk sl ‘"

other:
      remove
```

Figure 3-14: Sample Map File

The tag removal tool operates in a similar way to the tag replacement tool. A simple Lex/YACC parser is used to read the map file used by the tag replacement tool. A SED script is then created that removes the leader and trailer that were inserted into the code, restores special characters to their pre-replacement state, and removes the editor's tags from the file. The result is a straight text file that contains the source code after editing. Our system preserves the original form of the file as much as possible. No formatting or content is lost during either tag replacement or tag removal.

```
<MIFFile 3.00>
include(defaults.mif)
include(formatting.mif)
include(custom.mif)
include(leader.mif)
sp sl '#include <stdio.h\>' el ep
sp sl '' el ep
sp sl '' el CommentS sl '/*' el ep
sp sl ' * Simple function.' el ep
sp sl ' */' el CommentE sl '' el ep
sp sl '' el FuncHS sl 'void main()' el FuncHE sl ' {' el ep
sp sl '' el ep
sp sl '   int x;' el ep
sp sl '' el ep
sp sl '    ' el CommentS sl '/*' el ep
sp sl '     * Simple loop.' el ep
sp sl '     */' el CommentE sl '' el ep
sp sl '   for (x = 0; x < 3; x++) {' el ep
sp sl '' el ep
sp sl '      /*' el ep
sp sl '       * Simple if.' el ep
sp sl '       */ ' el ep
sp sl '      if (x == 2) {' el ep
sp sl '         fprintf(stderr, "X = 2\\n");' el ep
sp sl '      }' el ep
sp sl '   }' el ep
sp sl '}' el ep
include(trailer.mif)
```

Figure 3-15: Sample Source Code - With Editor Tags

The tag replacement and removal tools read the same map file that specifies how ITL tags correspond to editor tags. Both tools create SED scripts that perform the process of replacing the ITL tags with the editor's tags in the case of the tag replacement tool, or removing the editor's tags from the source code in the case of the tag removal tool.

3.7 The Editor

Once the source code has been parsed and tagged and the ITL tags replaced with editor-specific tags, the user can view and modify the embellished source code using the system's editor. The editor that was selected for use with the system is FrameMaker. Among its many features, FrameMaker allows the user to highlight text through the use of changes in font family and style, text size, and text colour. FrameMaker also provides facilities for using conditional text and greeking[4] text below a certain size, each of which can be used to provide a simple form of text elision. In addition, FrameMaker supports two tagging languages; Maker Markup Language (MML) and Maker Interchange Format (MIF) that can be used to tag text such as source code for later viewing and editing from within the editor. FrameMaker is discussed in more detail in Section 3.7.5.

---

[4]Text that is greeked is displayed as a thin grey rectangle.

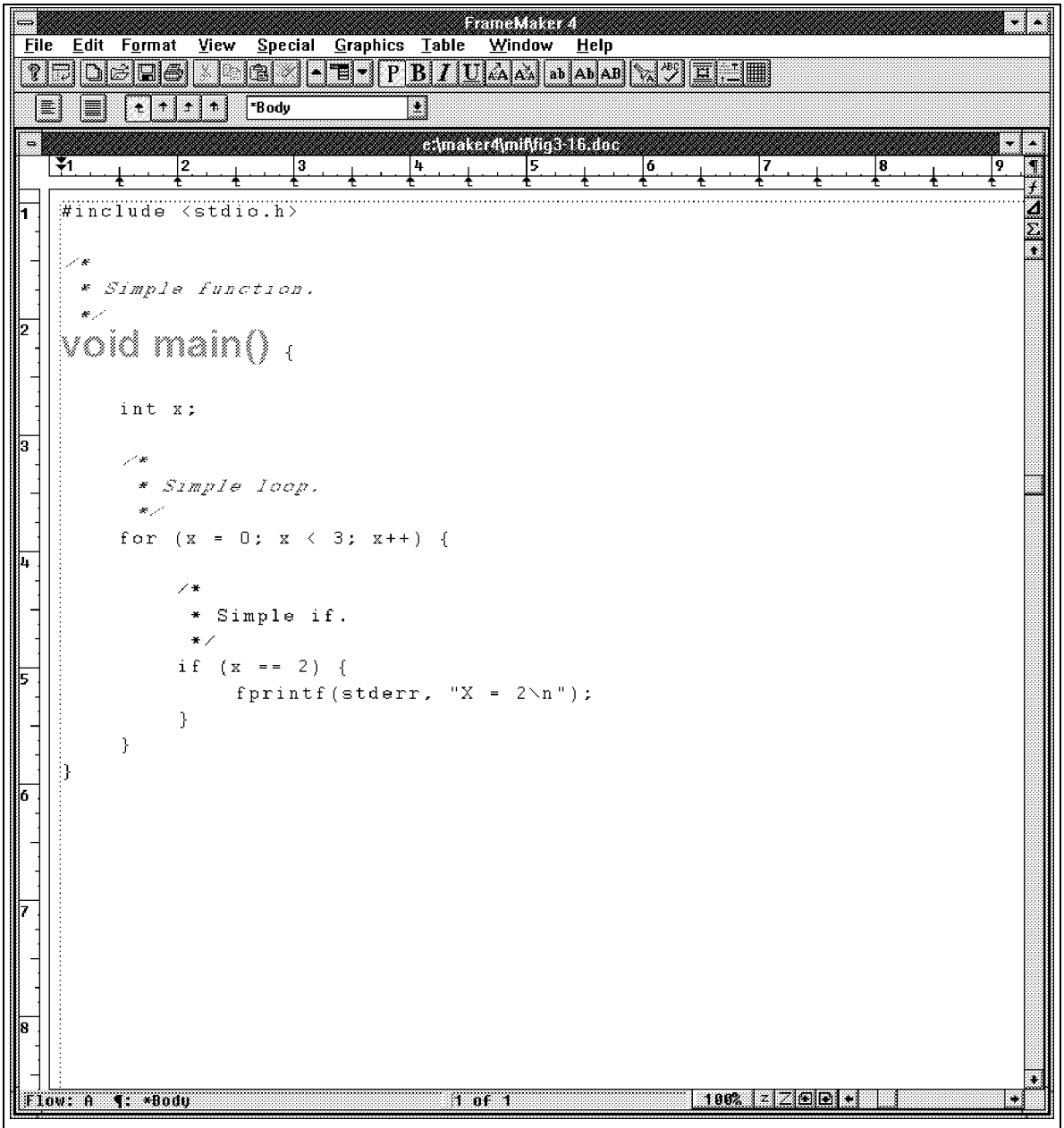

Figure 3-16: Sample Source Code - FrameMaker View

Figure 3-16 shows the file from Figure 3-15 as it might appear in FrameMaker. If we were to run the program in Figure 3-8 through a profiler, we might get the profiler data shown in Table 3-1. This profiler data could then be used to create the auxiliary data file shown in Figure 3-6. With the tag map file from Figure 3-14, our system could produce the editable view of the source code and profiler data that is shown in Figure 3-17.

Line Number	**Execution Time**	**Line Number**	**Execution Time**	**Line Number**	**Execution Time**
1	0.00	9	0.25	17	0.00
2	0.00	10	0.00	18	1.76
3	0.00	11	0.00	19	0.56
4	0.00	12	0.00	20	0.56
5	0.00	13	1.65	21	1.30
6	0.25	14	1.65	22	0.27
7	0.25	15	0.00		
8	0.25	16	0.00		

Table 3-1: Sample Profiler Data

Prior to selecting FrameMaker as the editor that would be used with the system, a number of other editing systems were examined and their suitability to this task evaluated. The editors that were examined were The Cornell Synthesizer Generator, IBM LPEX, SoftQuad Author/Editor, The University of Waterloo Computer Systems Group's Rita editor, and FrameMaker. The following sections discuss the advantages and disadvantages of each of the editors and analyzes each system's suitability to the task of viewing and editing a tagged source code document.

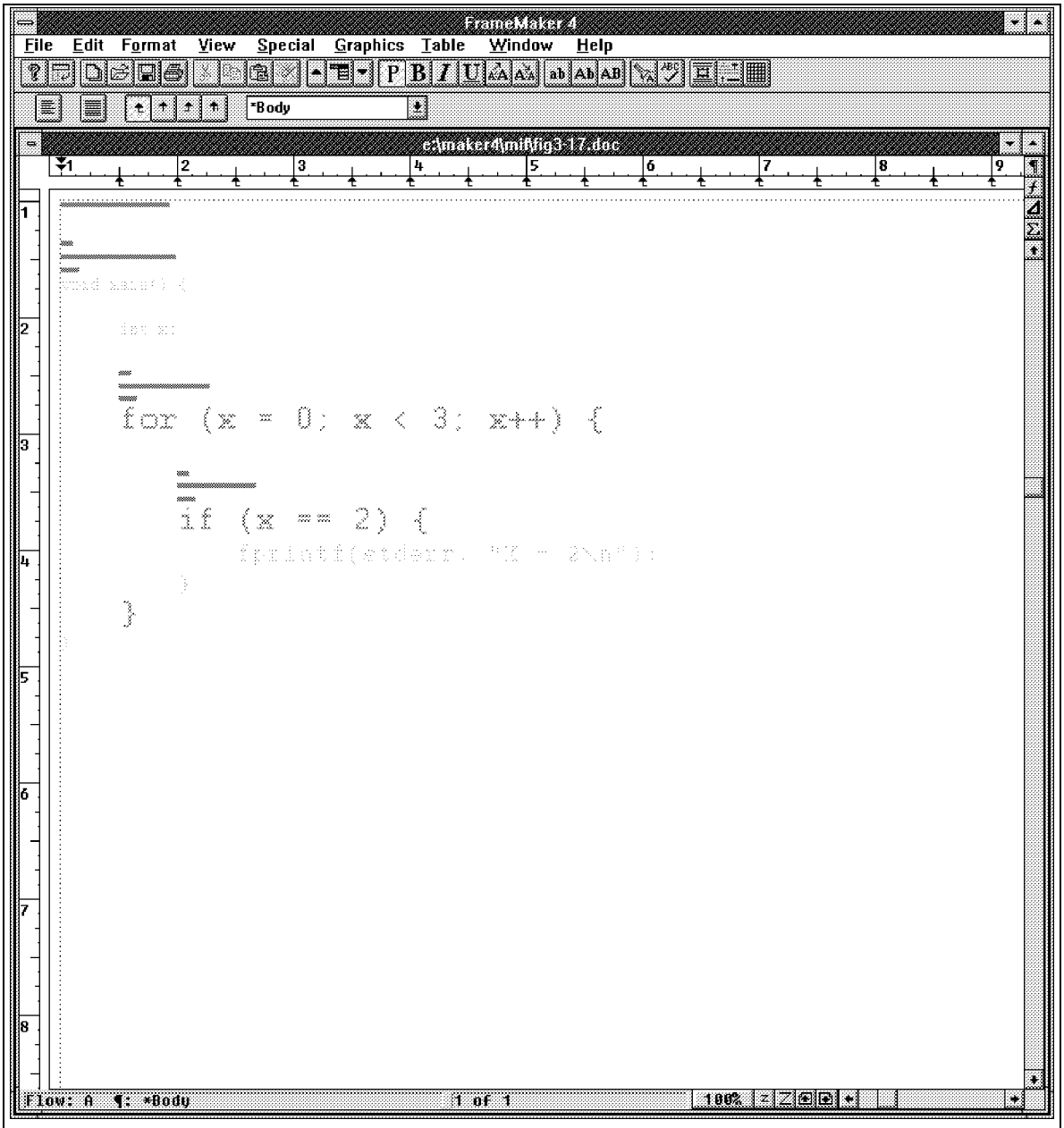

Figure 3-17: FrameMaker View Of Profiler Data

3.7.1 The Cornell Synthesizer Generator

The Cornell Synthesizer Generator [REPST88] is a system for creating language-based editing environments. Editors created with this tool continually use knowledge about the language specified for the editor to determine if a program contains errors and where these errors occur. The Synthesizer Generator uses an *immediate-computation* paradigm to perform analysis, translation, and error reporting while an object is being edited. Each change to a program triggers an immediate update of all affected analysis, error messages, and generated code. The Cornell Synthesizer Generator takes, as input, a specification of a language's abstract syntax, context-sensitive relationships, display format, concrete input syntax, and transformation rules for restructuring objects, and creates a display editor for manipulating objects according to these rules.

The Cornell Synthesizer Generator appears to provide the most power and flexibility of all the editors examined. It allows highlighting through changes in font family and style, text colour and size, as well as text elision. The Synthesizer Generator provides syntax checking and allows the use of arbitrary grammars. It is this generality, however, that makes the Cornell Synthesizer Generator unsuitable for this project. This tool provides more power and flexibility than is required for the editor for our system. In order to create an editor with the Synthesizer Generator, it would be necessary to create all of the language components (abstract syntax, etc.) for the C language. This overhead is one of the reasons that we decided to investigate the possibility of using other editors rather than

generating one with the Cornell Synthesizer Generator. In addition, the syntax is bound together with the editor. By using a separate parser, the system remains editor-independent.

3.7.2 LPEX

LPEX [IBMCO92] is the program editor used in the AIX SDE WorkBench/6000. WorkBench/6000 is a software development environment that runs on IBM RISC System/6000s running the Advanced Interactive Executive (AIX) operating system. The WorkBench is a collection of tools designed to assist in software development. The WorkBench toolset includes a program builder (compiler and linker), a program debugger, a static analyzer, and the programmable LPEX editor. LPEX provides standard text editing features such as text search and replacement, block move and copy, etc., and also allows the user to invoke a language-specific parser. The parser understands the language of the program being edited and can also display the structure of the source code using techniques such as colour, indentation, and font changes. LPEX provides parsers for C, C++, COBOL, and FORTRAN but the user can write parsers for other languages. The user can customize LPEX by writing macros to issue commands, adding frequently used commands to menus or creating new menus, and by building commands using the editor's application programming interface (API).

One advantage of LPEX over the other tools that were examined is that it understands the syntax of the C language. As a result, syntax highlighting can be provided automatically by LPEX. The system would then be responsible only for embellishments based on the auxiliary information. LPEX, while providing good reasons not to opt for editor independence, suffers from a couple of drawbacks that make it an unsuitable choice for the system's editor. The first drawback is that it is not possible to change the font size in LPEX because LPEX requires all fonts used to be the same point size as the default font. This is a significant limitation since changes in font size is one of the mechanisms for embellishing text that is required in this system. The second significant limitation of LPEX is that the LPEX C parser, when active, forces C syntax embellishment according to its own rules. Since the source code for this parser is not provided, the only way to preserve our own text formatting is to disable the parser. This is undesirable since by disabling the parser we would lose LPEX's automatic syntax highlighting. It is possible to write a custom parser for LPEX that could perform both syntax highlighting and embellishments based on auxiliary information tags in the source code, but this approach would require more time than was available.

3.7.3 SoftQuad Author/Editor

SoftQuad Author/Editor is a powerful multi-platform, Standard Generalized Markup Language (SGML), authoring system. SGML is an extensive text description language

that is quickly becoming the standard for text description. SGML provides a way in which anyone creating information can mark the components of that information; i.e. headings, captions, paragraphs, diagrams, and so on, using codes that can be understood by a wide variety of systems. SoftQuad Author/Editor provides word-processing features such as spell-checking, markup-sensitive search and replace, and keyboard shortcuts. It supports on-screen formatting by associating typographical features such as typeface and colour with structural elements. SoftQuad Author/Editor is structure-based and its automatic rules checking feature ensures correct markup during editing. The SGML parser provides a validation mechanism by checking the completeness of the entire SGML document.

A preliminary evaluation of the SoftQuad Author/Editor indicates that it has excellent potential to be the system's editor. It could easily be used with the system simply by setting up the proper tag replacement map file. However, as a commercial product of considerable expense, we could not afford it.

3.7.4 Rita

Rita [COWAN91] is a structured editor that ensures a document being edited conforms to a specified grammar or document type description (DTD). Rita guides the user in preparing a document by enforcing the correct placement of tags as specified by the DTD. Input to Rita is from a class database that describes the structure of the document (i.e. the document grammar or DTD) and the appearance of each structural

component as it appears on the screen (i.e. the style sheet). Output from Rita may be directed to different batch formatters and to different interactive displays.

Rita appears to be a good tool for ensuring that document structure is enforced, but its lack of on-screen text formatting is a serious limitation. Rita relies on back-end processing to format the document rather than providing built-in on-screen formatting. A separate formatting system is used to output a completed document to a display device such as a printer. Rita is not capable of displaying text in different font families, styles, or sizes or colours. Another significant drawback to Rita is that it is necessary to modify, compile, and load the DTD or style sheet for the document in order to change the grammar or appearance of a document. This involves significantly more effort than simply re-tagging a document. These two limitations make Rita unsuitable for use as the editor for the system. Rita, while capable of enforcing document structure, cannot provide embellished, editable views of source code.

3.7.5 FrameMaker

FrameMaker is a complete document composition, layout, and publishing package. It provides complete word-processing features such as text editing, text move and copy, spell checking, and text find and replace. Text can be formatted using changes in font family and style, text size, and text colour. FrameMaker allows you to use pre-defined paragraph and character formats to format text and to use cross references. You can

create *master* pages to simplify text layout by specifying the text and graphics that are to appear on each page of the document. FrameMaker allows you to create and use templates to maintain a consistent appearance from page to page, chapter to chapter, and document to document. In addition, FrameMaker has conditional text facilities that allow different versions of a document to be prepared in one FrameMaker file.

FrameMaker provides two tagging languages, Maker Markup Language (MML) and Maker Interchange Format (MIF), that can be used to create documents.

MML is used to create formatted FrameMaker documents from a text file. MML supports many of the formatting and layout features of FrameMaker. MML can be used to specify the content of a document for which formatting information is stored in a FrameMaker template or to specify both the content and format of a document. If the formatting information is stored in a template, only a small subset of MML instructions is required to specify when to use paragraph formats and when to change character formats for words and phrases. If a template is not used, then more complex MML statements are used to define formatting and layout specifications. MML appears to be a good tagging language for creating documents manually since it is straightforward and simple. The problem with MML is that it does not provide access to all of FrameMaker's formatting and layout features. For example, MML does not provide a mechanism for changing the colour of text and it does not provide support for conditional text.

MIF is a language that can be used to create an easily parsed, readable, text file containing all the text, graphics, formatting, and layout constructs that FrameMaker understands. MIF allows FrameMaker and other applications to exchange information while preserving graphics, document structure, and formatting. MIF is a more extensive and complex tagging language than MML so it is unsuitable for creating documents manually. MIF is designed so that filters can be written to convert automatically between MIF and other document formats. A significant benefit of MIF over MML is that MIF provides access to all of FrameMaker's features including changing the colour of text and conditional text. See Section 3.5.2 for more information on the MIF tagging language.

Of all the editing tools examined, FrameMaker seemed the most suitable for initial use as the system's editor. While the system we created is editor independent, we require an editor that provides sufficient display features (which eliminates LPEX and Rita) and that separates parsing from editing (which eliminates the Cornell Synthesizer Generator). These two factors were significant in selecting FrameMaker over the other tools. First, FrameMaker provides support for virtually all of the text formatting features that we require. It is possible to change the font family, style, text size and text colour in FrameMaker. While FrameMaker does not directly provide support for text elision, it does provide a conditional text facility as well as the ability to greek text below a certain size, either of which could be used to provide a simple form of text elision. Second, the MIF tagging language allows a source code file tagged using the Intermediate Tagging

Language to be easily converted into a document suitable for viewing and editing using FrameMaker.

Chapter 4

Sample Use

This chapter describes an example that shows how our system might be used to help a programmer understand a piece of source code.

We wrote the program shown in Figure 4-1 to count the number of characters and lines of text read from stdin. Figure 4-2 shows what this source code looks like in FrameMaker without any embellishments. Compiling this code and running it through a profiler might produce the profiler data shown in Table 4-1.

This profiler data is then converted into the auxiliary data file shown in Figure 4-3. This auxiliary data file along with the tag map file in Figure 3-14 is used to create the visualization of the program and profiler data shown in Figure 4-4.

Immediately, our attention is drawn to the section of source code in the large red font. This is code that counts the number of lines of text that has been read. Based on the map file that we used, we know that this is a section of code has been executed a lot. This

Line Number	Execution Time	Line Number	Execution Time	Line Number	Execution Time
1	0.00	12	0.00	23	0.00
2	0.00	13	0.00	24	0.00
3	0.00	14	0.00	25	0.00
4	0.00	15	1.76	26	0.00
5	0.00	16	0.00	27	1.40
6	0.00	17	0.00	28	0.00
7	0.25	18	0.00	29	0.00
8	0.25	19	0.00	30	1.40
9	0.30	20	1.80	31	0.00
10	0.25	21	1.95	32	0.30
11	0.00	22	1.95	33	0.25

Table 4-1: Line Count Program Profiler Data

seems odd because this block of code should not be executed any more frequently than the code that counts the number of characters that have been read. Also notice that the block of code that counts characters has been greeked, or reduced to a small grey rectangle. This indicates that this block of code has not been executed at all. Something must be wrong with the program. We then notice that our conditional statement

```
if (cChar = '\n') {
```

is actually an assignment statement that will always evaluate to true. We then correct the code as shown in Figure 4-5. Note that any code that is added assumes the font and colour attributes of the region where the addition is being made. After we have made the correction, we save the code without any tags so that we can re-compile it. Figure 4-6 shows the source code after it has been saved. Once we have re-compiled and re-run the

program through the profiler, we get the visualization shown in Figure 4-7. The code that counts the number of lines is now in a smaller magenta font while the code that counts the number of characters is in the large red font. This indicates that the line counting code block is executed less frequently than the character counting code block. This is more in-line with what we would expect from this type of program.

Although this is clearly a simple example, it demonstrates how our system can be used to present extrinsic information about a program in conjunction with the program's source code.

```
#include <stdio.h>

/*
 * This sample program counts the number of characters and lines of
 * text read from stdin.
 */
void main()
{
   int  nLines = 0, nChars = 0;
   char cChar;

   /*
    * Keep reading characters until we run out.
    */
   while ((cChar = getchar()) != EOF) {

      /*
       * Check for a new line.
       */
      if (cChar = '\n') {
         nLines++;
         nChars++;

      /*
       * Ordinary character.
       */
      } else {
         nChars++;
      }
   }

   fprintf(stderr, "Characters = %d\tLines = %d", nChars, nLines);
}
```

Figure 4-1: Line Count Program

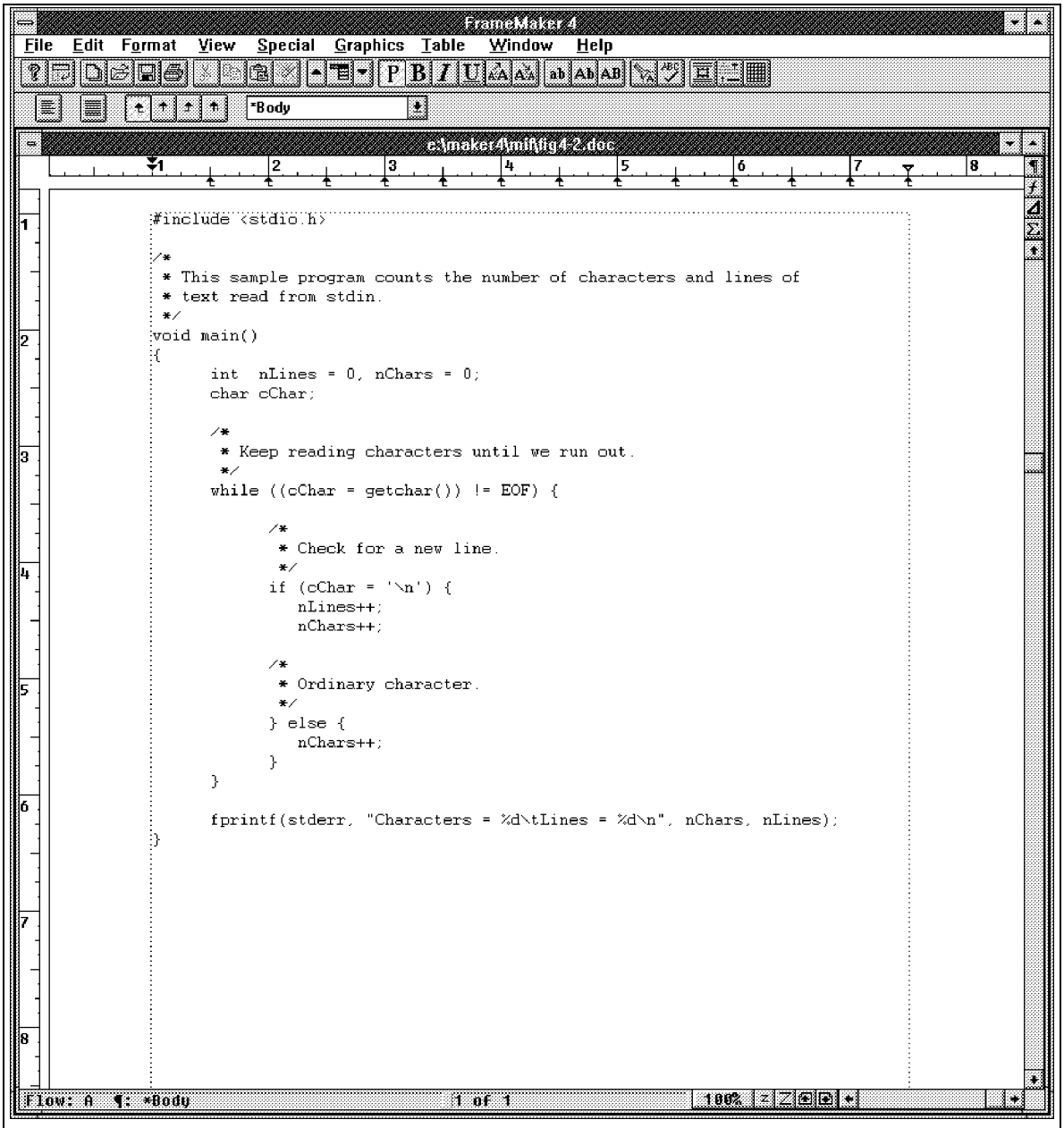

Figure 4-2: Line Count Program In FrameMaker

```
data:
      1 - 6             0.00
      7 - 8             0.25
      9                 0.30
      10                0.25
      11 - 14           0.00
      15                1.76
      16 - 19           0.00
      20                1.80
      21 - 22           1.95
      23 - 26           0.00
      27                1.40
      28 - 29           0.00
      30                1.40
      31                0.00
      32                0.30
      33                0.25

values:
      0.0               NONE_START        NONE_END
      0.1 - 0.49        LOW_START         LOW_END
      0.5 - 0.99        MEDIUM_START      MEDIUM_END
      1.0 - 2.0         HIGH_START        HIGH_END

labels:

priorities:
      lines             5
      comments          6
```

Figure 4-3: Auxiliary Data File For Line Count Program Profiler Data

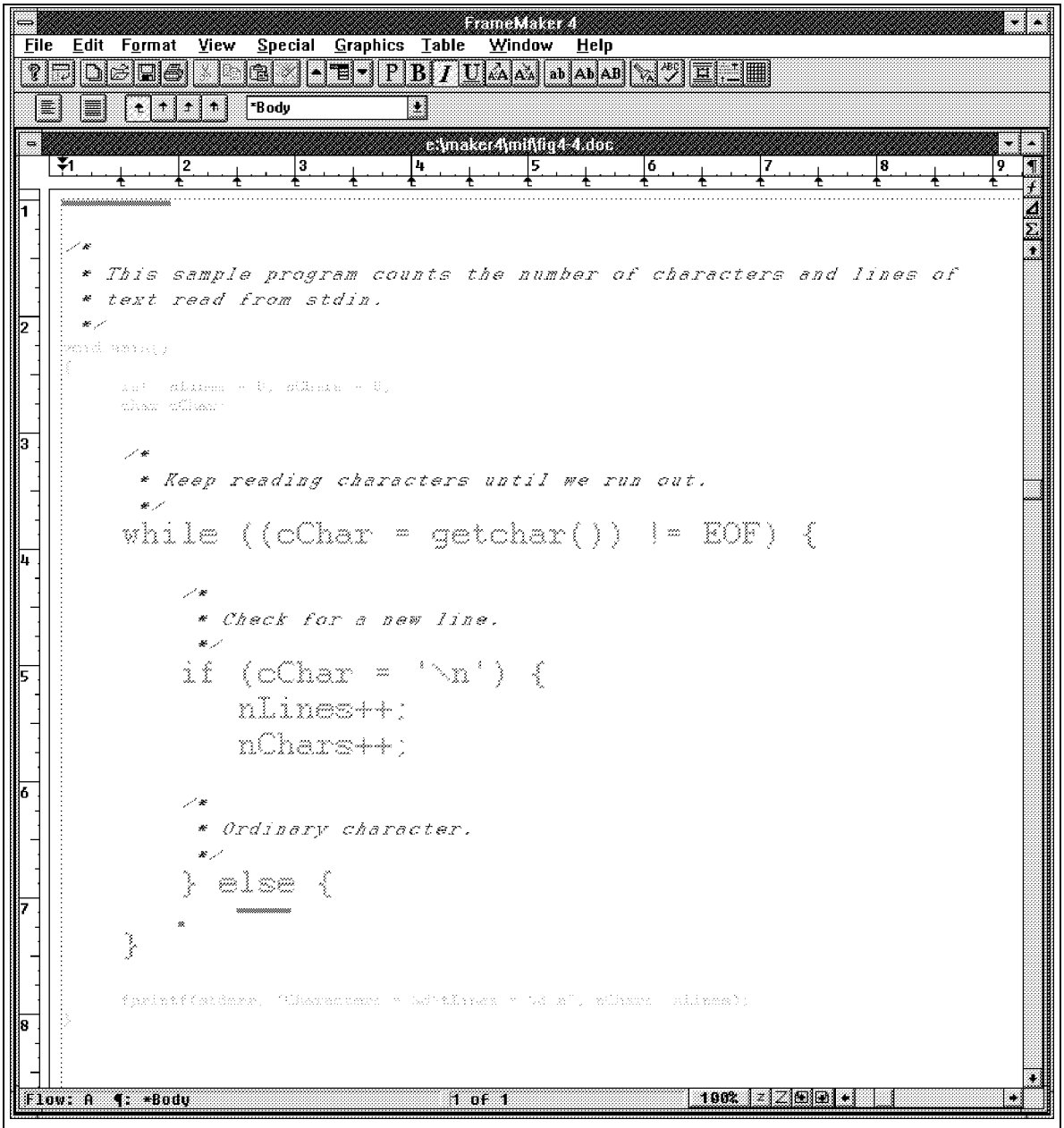

Figure 4-4: Visualization Of Line Count Program Profiler Data

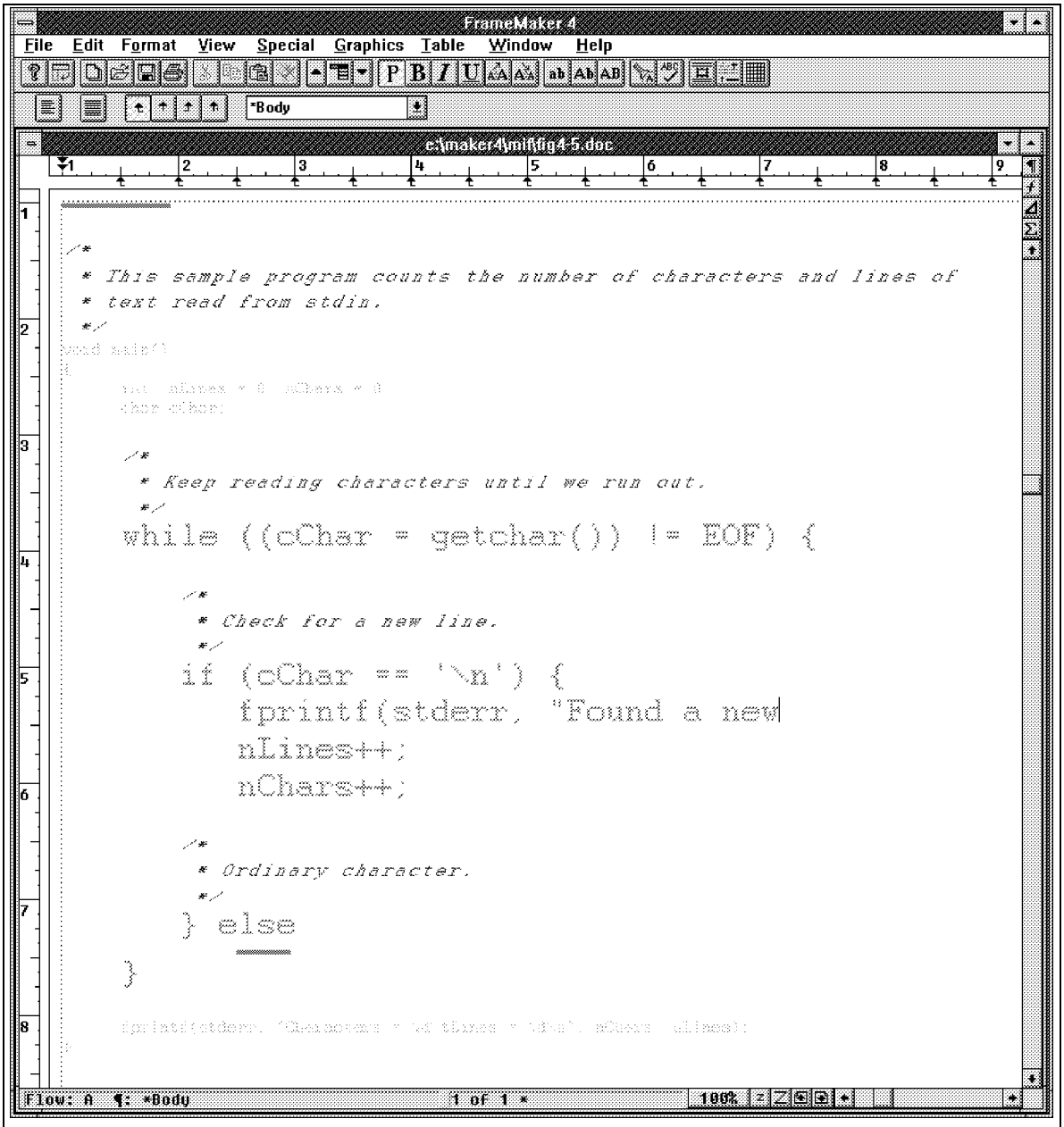

Figure 4-5: Modifying The Line Count Program

```
#include <stdio.h>

/*
 * This sample program counts the number of characters and lines of
 * text read from stdin.
 */
void main()
{
   int  nLines = 0, nChars = 0;
   char cChar;

   /*
    * Keep reading characters until we run out.
    */
   while ((cChar = getchar()) != EOF) {

      /*
       * Check for a new line.
       */
      if (cChar == '\n') {
         fprintf(stderr, "Found a newline\n");
         nLines++;
         nChars++;

      /*
       * Ordinary character.
       */
      } else
         nChars++;
   }

   fprintf(stderr, "Characters = %d\tLines = %d", nChars, nLines);
}
```

Figure 4-6: Modified Line Count Program

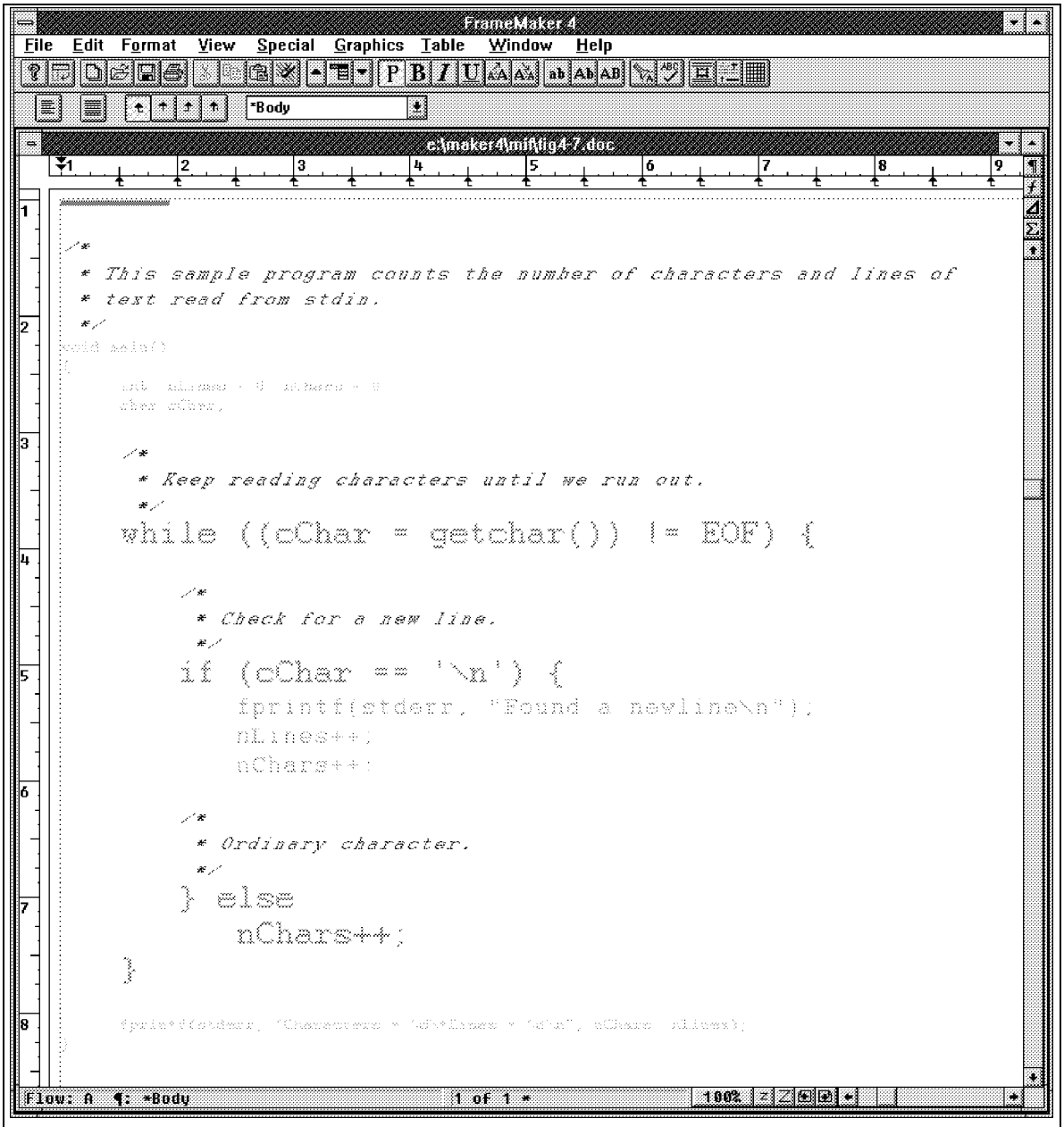

Figure 4-7: Visualization Of Modified Line Count Program Profiler Data

Chapter 5

Summary And Conclusions

The purpose of this research project was to design and develop a prototype system that would assist software developers and maintainers in understanding a program's source code. To accomplish this, we use changes in font size, style, and family and text colour to present extrinsic information about the program in editable source code views. A review of current literature indicates that very little work has been done in the area of improving program understanding. Particularly surprising is the lack of research in providing editable views of source code. Understanding software is a very time consuming and hence expensive process and more research is required to develop tools that can simplify this process. Although our system represents an important step in the development of tools to assist in software understanding, it is clear that more effort must be directed towards this area of research.

Part of the motivation for this thesis was to prove that it is possible to present extrinsic information about a program in editable views. Our system addresses some of the limitations in existing software development and maintenance tools. One of the key issues is the ability to provide multiple, editable views of the source code to the user. Much of the existing work has focused on single, static, non-editable views of code. Editable views provide a consistent, natural environment in which to do software development and maintenance because they alleviate the need to switch applications when viewing information and when making changes based on this information. Our system can provide multiple, editable views of source code.

A number of interesting issues arose during the development of the system. One such issue related to editable views is the difficulty in presenting relevant information in a useful manner within an editable view. Textual embellishment is a simple, yet effective mechanism for conveying certain types of information. However, preserving the embellishments during editing is complicated from an implementation as well as from a semantic perspective. In our system, embellishments are performed on lines of source code and on syntactic elements of the programming language. As the user modifies the source code within an editable view, lines and syntactic elements are added, deleted, and become fragmented. As a result, it is difficult for the editor to preserve the essence of the original formatting. In addition, it is unclear what meaning the embellishments have as the source code is modified. Syntactic highlighting can remain meaningful in the presence of

modifications provided the editor continually parses the code and updates the highlighting. However, the line-oriented embellishments can become meaningless and useless as a file is modified. The issues become even more complicated as different forms of embellishments are used for more complex information. This is a difficult and complicated issue that requires more study before it can be properly addressed.

The method of presenting information to the user is another challenging issue that arose during the development of the system. We decided to present information using textual embellishments. Our system uses changes in font size, style, family and text colour to present information to the user. This approach was taken because it was simple to implement, since FrameMaker makes it easy to change these text attributes using MIF. In addition, we felt that textual embellishments would be an effective mechanism for presenting the line-oriented information towards which our system is geared. MIF provides support for all of the textual changes that we required, however, MIF and FrameMaker do not provide support for automatic text elision. As a result, we were forced to simulate elision using very small font sizes in conjunction with greeking or with the conditional text facility of FrameMaker. It would be interesting to extend the system to present information using different techniques depending on the type of information. Possibilities include the use of icons and other pictures, sound, animation, and more. Alternative methods of presenting information and determining how to best present information are areas that require further research.

The need to build a system that could accept information from a wide variety of data sources posed some interesting challenges. Clearly, no solution could be complete since the number and types of information is overwhelmingly large. The goal was to come up with a strategy that would encompass as broad a set of data sources as possible. We decided to concentrate on line-oriented data because of the large number of tools that produce this type of information. The auxiliary data file was designed to be a common repository for line-oriented data. Any tool that produces line-oriented information can be used with our system simply by converting the information output by the tool into an auxiliary data file. This process of data conversion can be automated in most cases. The decision was made to use an intermediate file to keep the system open and extensible. It would have been impossible to develop a system that could accept input directly from any tool that produced line-oriented data. The auxiliary data file allows us to accept data from any tool that produces line-oriented information. Although an intermediate data file allows the use of a broad range of external tools, there are still a large number of applications that do not produce line-oriented data and therefore cannot be used with the system. Extending the system to accept information from other, non-line-oriented data sources would be an interesting exercise. This would improve the usability of the system as it would then appeal to a broader range of users, but it would involve a significant amount of additional work.

The final major requirement of the system was that it be extensible. The modular

design of the system provides some extensibility. If the user desires different functionality in a particular component of the system, that component can be replaced with one that performs as desired. One of the most significant examples of the extensibility of the system is its editor independence. We use a custom tagging language for the intermediate markup of the source code files and then replace these tags with editor-specific tags to keep the system flexible. The only requirement on the editor used with the system is that it have an accessible, ASCII-tagged storage format that can be mapped onto the Intermediate Tagging Language. Different editors can be used simply by creating different tag replacement map files for each editor. One of the drawbacks in the design of the system is that it is not as modular as it could be. The tagging module is tightly coupled with the parser in the system. As a result, modifying the system to use a different parser would require significant modifications to the tagging module. Ideally, the interface between the tagging module and the parser would be simple and straightforward. A different parser could then be written that conformed to this interface and be easily inserted into the system. This would allow the system to be easily adapted to be used with different source code languages. Another modification that would improve the extensibility of the system would be the re-implementation of the system in C++. An object-oriented design and implementation of the system would facilitate a more extensible system. For example, the parser could simply be an object in the system that

communicated to the tagging object through a well-defined series of methods. It would then be fairly straightforward to interchange objects in the system.

This research project has proven that an extensible system can be developed that presents extrinsic information from a variety of sources in conjunction with source code in editable views. What has not yet been proven, at least empirically, is that such a system can improve software developer and maintainer productivity. Tapp showed that embellishing source code using colour helps programmers understand the code better then when no embellishments are used [TAPPR94]. Tapp's results when embellishing using font changes were inconclusive, but suggest that they could have some positive effect on program understanding. Since our system creates editable views by embellishing source code using colour and font changes, we anticipate that our system can help programmers understand source code better and thereby improve their productivity. However, usability tests should be designed and performed to confirm the benefits of such an editable views system.

In conclusion, we have designed an extensible system that presents extrinsic information about source code in editable views. The views are created by embellishing the source code using changes in font family, size, and style, text colour, and text elision. We believe that such a system can improve software developer and maintainer productivity by assisting in understanding source code. Empirical usability tests are required to confirm or deny this hypothesis.

Appendix A

Auxiliary Data File Specification

This appendix gives the complete specification for the auxiliary data file. The specification is given in grammar form with the following conventions:

- Text contained in angle brackets (<>) represents a non-terminal of the grammar (e.g. <value list>).
- Italicized text is a description of one or more terminals in the grammar (e.g. *the newline character*).
- Normal text represents the text of a terminal in the grammar (e.g. data:).
- Each rule in the grammar is of the form **<A> ::= exp1 | exp2 | ...** where <A> is a non-terminal in the grammar and exp1, exp2, etc., are expansions of the non-terminal <A> consisting of a series of terminals and/or non-terminals. This rule means that <A> can expand to exp1 or exp2 or ...

The grammar for the auxiliary data file is given in the following pages. Figure A-1 shows an annotated example of a complete auxiliary data file.

<auxiliary data file> ::= <data section[5]><NL>

[5]The data section assigns values or labels to lines or identifiers in the source code.

<value mapping section[6]><NL>
<label mapping section[7]><NL>
<priorities section[8]>

<NL> ::= *the newline character*

<SEP> ::= *a sequence of spaces or tabs*

<data section> ::= data:<NL><data entries> | data:<NL>*nothing*

<data entries> ::= <data entry> | <data entry><NL><data entries>

<data entry> ::= <data><SEP><value> | <data><SEP><label>

<data> ::= <value list> | <identifier list>

<value list> ::= <value range> | <value range><SEP>,<SEP><value list>
(e.g. 6 and 3.1 - 3.5 , 7 are both valid)

<identifier list> ::= <identifier> | <identifier><SEP>,<SEP><identifier list>
(e.g. for and foo , bar are both valid)

<value range> ::= <value> | <value><SEP>-<SEP><value>
(e.g. 10.4 and 8 - 11 are both valid)

<value> ::= *some integer or floating point value (e.g. 12.4 or 6)*

<label> ::= *a sequence of printable ASCII characters containing no spaces (e.g. foo_bar)*

<identifier> ::= *a legal C identifier (e.g. nMyValue)*

---

[6]The value mapping section defines how values in the data section are mapped to tags.

[7]The label mapping section defines how labels in the data section are mapped to tag.

[8]The priorities section defines the priorities of constructs which are used when tagging the source code using prioritized, stack-based tagging.

<value mapping section> ::= values:<NL><value entries> | values:<NL>*nothing*

<value entries> ::= <value entry> | <value entry><NL><value entries>

<value entry> ::= <value list><SEP><tag text><SEP><tag text>

<tag text> ::= *a sequence of printable ASCII characters containing no spaces (e.g. FOO_START) - these sequences will be surrounded by %[and %] when tagging the source code (e.g. %[FOO_START%])*

<label mapping section> ::= labels:<NL><label entries> | labels:<NL>*nothing*

<label entries> ::= <label entry> | <label entry><NL><label entries>

<label entry> ::= <label list><SEP><tag text><SEP><tag text>

<priorities section> ::= priorities:<NL><priority entries> | priorities:<NL>*nothing*

<priority entries> ::= <priority entry> | <priority entry><NL><priority entries>

<priority entry> ::= <construct label><SEP><priority>

<priority> ::= *some integer value (e.g. 3 or -1)*

<construct label> ::= strings | fcalls | typedefs | declarations | compounds | compound_bodies | ifs | if_headers | if_bodies | switches | switch_bodies | whiles | while_headers | while_bodies | dos | do_headers | do_bodies | fors | for_headers | for_bodies | gotos | continues | breaks | returns | function_headers | identifiers | comments | lines | jumps | loops | loop_headers | loop_bodies | conditionals | conditional_headers | conditional_bodies | definitions

```
data:
      1 - 5       FileStart   <- This section assigns values and/or
      6 - 9       0.25           labels to various lines and/or
      10 - 12     0.0            dentifiers in the source code file.
      13 - 14     1.65           In this example, the value 1.65 has
      15 - 17     0.0            been associated with the lines 13 and
      18          1.76           14 and the label Boolean has been
      19 - 20     FileEnd        associated with the identifier bFlag.
      nIndex      Integer
      bFlag       Boolean

values:
      0.0         NONE_START  NONE_END    <- This section defines how
      0.1 - 0.49  LOW_START   LOW_END        values in the data section
      0.5 - 0.99  MED_START   MED_END        map to tags. In this
      1.0 - 2.0   HIGH_START  HIGH_END       example, lines or
                                             identifiers that have a
                              value between 0.1 and 0.49 associated
                              with them will be surrounded by the tags
                              %[LOW_START%] and %[LOW_END%].

labels:
      FileStart   FS_START    FS_END      <- This section defines how
      FileEnd     FE_START    FE_END         labels in the data section
      Integer     INT_START   INT_END        map to tags.  In this
      Boolean     BOOL_START  BOOL_END       example, lines or
                                             identifiers that have a
                              label of Integer associated with them will
                              be surrounded by the tags %[INT_START%]
                              and %[INT_END%]

priorities:
      lines             5     <- This section defines the priorities
      if_bodies         4        of constructs which are used when
      if_headers        -1       tagging the source code using
                                    prioritized, stack-based tagging.
                              In this example, lines will be tagged
                              with priority 5 and the headers of if
                              statements with priority -1.
```

Figure A-1: Annotated Auxiliary Data File

Appendix B

Tag Map File Specification

This appendix gives the complete specification for the tag map file. The specification is given in grammar form with the following conventions:

- Text contained in angle brackets (<>) represents a non-terminal of the grammar (e.g. <replacement text>).
- Italicized text is a description of one or more terminals in the grammar (e.g. *the newline character*).
- Normal text represents the text of a terminal in the grammar (e.g. pass1:).
- Each rule in the grammar is of the form **<A> ::= exp1 | exp2 | ...** where <A> is a non-terminal in the grammar and exp1, exp2, etc., are expansions of the non-terminal <A> consisting of a series of terminals and/or non-terminals. This rule means that <A> can expand to exp1 or exp2 or ...

The grammar for the tag map file is given in the following pages. Figure B-1 shows an annotated example of a complete tag map file.

<tag map file> ::= <leader section[9]><NL>
<trailer section[10]><NL>
<pass1 section[11]><NL>
<pass2 section[12]><NL>
<other section[13]>

<NL> ::= *the newline character*

<SEP> ::= *a sequence of spaces or tabs*

<leader section> ::= leader:<NL><leader or trailer entries> | leader:<NL>*nothing*

<leader or trailer entries> ::= <leader or trailer entry> |
<leader or trailer entry><NL><leader or trailer entries>

<leader or trailer entry> ::= <quoted text>

<quoted text> ::= "*some sequence of printable ASCII characters*"
(e.g. "include (defaults.mif)")

<trailer section> ::= trailer:<NL><leader or trailer entries> | trailer:<NL>*nothing*

---

[9]The leader section identifies text that will be inserted at the beginning of the mapped file.

[10]The trailer section identifies text that will be inserted at the end of the mapped file.

[11]The pass1 section identifies text replacements that should be done on the first pass through the file. Typically these replacements are special character replacements such as replacing tab characters with an escape sequence or the start of each line with a special sequence of characters.

[12]The pass2 section identifies text replacements that should be done on the second pass through the file. Typically these replacements are tag replacements; replacing system or user-defined tags with editor-specific tag sequences.

[13]This section defines what should be done with unhandled tags - whether they should be left in the text or removed.

<pass1 section> ::= pass1:<NL><pass1 entries> | pass1:<NL>*nothing*

<pass1 entries> ::= <pass1 entry> | <pass1 entry><NL><pass1 entries>

<pass1 entry> ::= <token><SEP><replacement text>
(e.g. start "sp sl ‘" and ">" "\>" are both valid)

<token> ::= <quoted text> | start | end | tab | vtab | nline | ffeed

<replacement text> ::= <quoted text>

<pass2 section> ::= pass2:<NL><pass2 entries> | pass2:<NL>*nothing*

<pass2 entries> ::= <pass2 entry> | <pass2 entry><NL><pass2 entries>

<pass2 entry> ::= <construct tag><SEP><replacement text> |
<quoted text><SEP><replacement text>
(e.g. if_body_start "' el IfBS sl `" and "my_tag" "' el MyTag sl `" are both valid)

<construct tag> ::= string_start | string_end | fcall_start | fcall_end | typedef_start |
typedef_end | declaration_start | declaration_end | compound_start |
compound_end | compound_body_start | compound_body_end | if_start |
if_end | if_header_start | if_header_end if_body_start | if _body_end |
switch_start | switch_end | switch_header_start | switch_header_end |
switch_body_start | switch_body_end | while_start | while_end |
while_header_start | while_header_end | while_body_start |
while_body_end | do_start | do_end | do_header_start | do_header_end |
do_body_start | do_body_end | for_start | for_end | for_header_start |
for_header_end | for_body_start | for_body_end | goto_start | goto_end |
continue_start | continue_end | break_start | break_end | return_start |
return_end | function_header_start | function_header_end | identifier_start |
identifier_end | comment_start | comment_end | jump_start | jump_end |
loop_start | loop_end | loop_header_start | loop_header_end |
loop_body_start | loop_body_end | conditional_start | conditional_end |
conditional_header_start | conditional_header_end |
conditional_body_start | conditional_body_end | definition_start |
definition_end

<other section> ::= other:<NL><other entry> | other:<NL>*nothing*

<other entry> ::= remove | leave

```
leader:
      "<MIFFile 3.00>"              <- This section specifies text that
      "include(defaults.mif)"          will be inserted at the beginning
      "include(formatting.mif)"        of the mapped file.  In this
      "include(custom.mif)"            example, each of these lines of
      "include(leader.mif)"            text will be inserted (without
                                       the quotes).

trailer:
      "include(trailer.mif)"        <- This section specifies text that
                                       will be inserted at the end of
                                       the mapped file.

pass1:
      start "sp sl ‘"          <- This section defines special text
      end   "’ el ep"             replacements that will be done on the
      "\"   "\\"                  first pass through the tagged file.
      "’"   "\q"                  These are typically special character
      "‘"   "\Q"                  replacements.  In this example, the
      ">"   "\>"                  start of each line will be replaced
      tab   "\t"                  with the character sequence sp sl ‘.

pass2:
      if_header_start   "’ el IfHS sl ‘"  <- This section defines how
      if_header_end     "’ el IfHE sl ‘"     tags will be replaced on
      if_body_start     "’ el IfBS sl ‘"     the second pass through the
      if_body_end       "’ el IfBE sl ‘"     tagged file.  In this
      "LOW_START"       "’ el LowS sl ‘"     example, the system tag for
      "LOW_END"         "’ el LowE sl ‘"     the start of if body
                                             headers will be replaced
                              with the text "’ el IfBS sl ‘" and the
                              user-defined tag %[LOW_END%] will be
                              replaced with the text "’ el LowE sl ‘".

other:
      remove            <- This section determines what is done with
                           tags that are not replaced in the pass2
                           section.  In this example, tags that are
                           not replaced are removed.
```

Figure B-1: Annotated Tag Map File

Appendix C

Intermediate Tagging Language Specification

Each tag in ITL is of the form *%[<tag text>%]* where *<tag text>* is some sequence of printable ASCII characters containing no spaces (e.g. MYTAG_START). Every construct that is tagged is surrounded by a pair of tags; a start tag and an end tag. For example, if we are tagging an identifier called *foo*, we may define the start tag text for *foo* to be FOO_START and the end tag text to be FOO_END. The start and end tags would then be %[FOO_START%] and %[FOO_END%] respectively.

The user can define custom tags within the auxiliary data file for tagging lines of source code or identifiers. Table C-1 lists the constructs that are automatically tagged by the system with their identifiers from the auxiliary data file and the tag map file. The auxiliary data file identifiers are used in the *priorities* section of the auxiliary data file to assign priorities to the constructs when they are tagged. The tag map file identifiers are

used in the *pass2* section of the tag map file to identify the tags to be replaced. Table C-2 lists the ITL tags that correspond to each auxiliary data file identifier.

Construct	Auxiliary Data File Identifier	Tag Map File Start Tag Identifier	Tag Map File End Tag Identifier
strings	strings	string_start	string_end
function calls	fcalls	fcall_start	fcall_end
type definitions	typedefs	typedef_start	typedef_end
declaration statements	declarations	declaration_start	declaration_end
compound statements	compounds compound_bodies	compound_start compound_body_start	compound_end compound_body_end
if statements	ifs if_headers if_bodies	if_start if_header_start if_body_start	if_end if_header_end if_body_end
switch statements	switches switch_headers switch_bodies	switch_start switch_header_start switch_body_start	switch_end switch_header_end switch_body_end
while statements	whiles while_headers while_bodies	while_start while_header_start while_body_start	while_end while_header_end while_body_end
do statements	dos do_headers do_bodies	do_start do_header_start do_body_start	do_end do_header_end do_body_end
for statements	fors for_headers for_bodies	for_start for_header_start for_body_start	for_end for_header_end for_body_end
gotos	gotos	goto_start	goto_end
continues	continues	continue_start	continue_end
breaks	breaks	break_start	break_end
returns	returns	return_start	return_end
function headers	function_headers	function_header_start	function_header_end
comments	comments	comment_start	comment_end
jumps (goto, continue,	jumps	jump_start	jump_end

Construct	Auxiliary Data File Identifier	Tag Map File Start Tag Identifier	Tag Map File End Tag Identifier
break, return)			
loops (for, do, while)	loops loop_headers loop_bodies	loop_start loop_header_start loop_body_start	loop_end loop_header_end loop_body_end
conditionals (if, switch)	conditionals conditional_headers conditional_bodies	conditional_start conditional_header_start conditional_body_start	conditional_end conditional_header_end conditional_body_end
definitions (declarations, typedefs)	definitions	definition_start	definition_end

Table C-1: Auxiliary Data And Tag Map File Identifiers

Auxiliary Data File Identifier	Start Tag	End Tag
strings	%[STRING_START%]	%[STRING_END%]
fcalls	%[FCALL_START%]	%[FCALL_END%]
typedefs	%[TDEF_START%]	%[TDEF_END%]
declarations	%[DECL_START%]	%[DECL_END%]
compounds compound_bodies	%[CMPND_START%] %[CB_START%]	%[CMPND_END%] %[CB_END%]
ifs if_headers if_bodies	%[IF_START%] %[IH_START%] %[IB_START%]	%[IF_END%] %[IH_END%] %[IB_END%]
switches switch_headers switch_bodies	%[SWITCH_START%] %[SH_START%] %[SB_START%]	%[SWITCH_END%] %[SH_END%] %[SB_END%]
whiles while_headers while_bodies	%[WHILE_START%] %[WH_START%] %[WB_START%]	%[WHILE_END%] %[WH_END%] %[WB_END%]
dos do_headers do_bodies	%[DO_START%] %[DH_START%] %[DB_START%]	%[DO_END%] %[DH_END%] %[DB_END%]
fors	%[FOR_START%]	%[FOR_END%]

Auxiliary Data File Identifier	Start Tag	End Tag
for_headers for_bodies	%[FH_START%] %[FB_START%]	%[FH_END%] %[FB_END%]
gotos	%[GOTO_START%]	%[GOTO_END%]
continues	%[CONTINUE_START%]	%[CONTINUE_END%]
breaks	%[BREAK_START%]	%[BREAK_END%]
returns	%[RETURN_START%]	%[RETURN_END%]
function_headers	%[FHEADER_START%]	%[FHEADER_END%]
comments	%[COMMENT_START%]	%[COMMENT_END%]
jumps	the start tags for gotos, continues, breaks, and returns	the end tags for gotos, continues, breaks, and returns
loops loop_headers loop_bodies	the start tags for fors, dos, and whiles)	the end tags for fors, do, and whiles)
conditionals conditional_headers conditional_bodies	the start tags for ifs, if_headers, if_bodies, switches, switch_headers, and switch_bodies	the end tags for ifs, if_headers, if_bodies, switches, switch_headers, and switch_bodies
definitions	the start tags for declarations and typedefs	the end tags for declarations and typedefs

Table C-2: ITL Start And End Tags

Bibliography

[BENBA86] Benbasat, I., Dexter, A. S., and Todd, P., (1986), "The Influence of Color and Graphical Information Presentation in a Managerial Decision Simulation", *Human-Computer Interaction*, 2, 65-92.

[BENZOXX] Benzon, W., "Visual Thinking", Bibliographic Information Unknown, 411-427.

[BOCKE90] Böcker, H. and Herczeg, J., (1990), "What Tracers are Made of", *ACM Sigplan Notices*, 25(10), 89-99.

[BRODN93] Brodnik, A. and Tompa, F. W., (1993), "A New Architecture to Support Database Updates Through Views", Unpublished Paper, Department of Computer Science, University of Waterloo, Waterloo, Ontario.

[BROWN84] Brown, M. H. and Sedgewick, R., (1984), "A System for Algorithm Animation", *ACM Transactions on Computer Graphics*, 18(3), 177-186.

[CARDS91] Card, S. K., Robertson, G. G., and Mackinlay, J. D., (1991), "The Information Visualizer, an Information Workspace", In *Proceedings CHI '91 Conference on Human Factors in Computing Systems*, Addison-Wesley, Reading, Massachusetts, 181-188.

[CHENY86] Chen, Y. and Ramamoorthy, C. V., (1986), "The C Information Abstractor", In *Proceedings COMPSAC '86 International Computer Software and Applications Conference*, IEEE Computer Society Press, New York, New York, 291-298.

[CLARK89] Clark, G., Clark, J., and Ling, T. W., (1989), "Addressing the Problem of Software Library Management for Object Oriented and Conventional Operator/Operand Systems, Utilizing the Entity-Relationship Approach", *Database*, 20(1), 30-35.

[CLEVE89] Cleveland, L., (1989), "A Program Understanding Support Environment", *IBM Systems Journal*, 28(2), 324-344.

[COLLO85] Collofello, J. S. and Blaylock, J. W., (1985), "Syntactic Information Useful for Software Maintenance", In *Proceedings National Computer Conference*, American Federation of Information Processing Societies Press, Montvale, New Jersey, 547-553.

[CORBI89] Corbi, T. A., (1989), “Program understanding: Challenge for the 1990s”, *IBM Systems Journal*, 28(2), 294-306.

[CORDY90] Cordy, J. R., (1990), "TuringTool: A User Interface to Aid in the Software Maintenance Task", *IEEE Transactions on Software Engineering*, 16(3), 294-301.

[COWAN90] Cowan, D. D., Mackie, E. W., and Pianosi, G. M., (1990), "Experience with RITA - A Structured-Document Editor", Research Report CS-90-35, Department of Computer Science, University of Waterloo, Waterloo, Ontario.

[COWAN91] Cowan, D. D., Mackie, E. W., Pianosi, G. M., and Smit, G. de V., (1991), "Rita - An Editor and User Interface for Manipulating Structured Documents", *Electronic Publishing*, 4(3), 125-150.

[CULIK92] Culik, C., (1992), *View Updates in Relational Databases*, M. Math Thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario.

[CYBUL92] Cybulski, J. L and Reed, K., (1992), "A Hypertext Based Software-Engineering Environment", *IEEE Software*, 9(2), 62-68.

[DESAN84] DeSanctis, G., (1984), "Computer Graphics as Decision Aids: Directions and Research", *Decision Sciences*, 15(4), 463-487.

[DIXAL93] Dix, A., Finlay, J., Abowd, G., and Beale, R., (1993), *Human-Computer Interaction*, Prentice Hall, London, England.

[EICKS92] Eick, S. G., Steffen, J. L., and Sumner, E. E., (1992), "Seesoft - A Tool For Visualizing Software Statistics", *IEEE Transactions on Software Engineering*, 18(11), 957-968.

[EVANS90] Evans, R. J., (1990), "SMARTsystem: A CASE Development Environment for Existing C Programs", In *Proceedings 1990 IEEE Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, California, 256.

[FRAME92a] Frame Technology Corporation, (1992), *MIF Reference*, Frame Technology Corporation, San Jose, California

[FRAME92b] Frame Technology Corporation, (1992), *Using FrameMaker*, Frame Technology Corporation, San Jose, California

[FURNA86] Furnas, G. W., (1986), “Generalized Fisheye Views”, In *Proceedings CHI ‘86*, ACM Press, New York, New York, 16-23.

[GURAR91] Gurari, E. M. And Wu, J., (1991), “A WYSIWYG Literate Programming System (Preliminary Report)”, In *Proceedings ACM Computer Science Conference*, ACM Press, New York, New York, 94-104.

[HARBA90] Harband, J., (1990), "SEELA: Maintenance and Documenting by Reverse-Engineering", In *Proceedings IEEE Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, California, 146.

[HELMR91] Helm, R. and Maarek, Y. S., (1991), "Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries", *ACM Sigplan Notices*, 26(11), 47-61.

[HUDLI89] Hudlicka, E., (1989), "Visual System Browser", *ACM SIGCHI Bulletin*, 20(4), 18-24.

[IBMCO92] IBM Corporation, (1992), *IBM AIX SDE WorkBench/6000 Program Editor 2nd Edition*, IBM Corporation, North York, Ontario

[JONES62] Jones, M. R., (1962), "Color Coding", *Human Factors*, 4(6), 355-365.

[LOVET77] Love, T., (1977), "An Experimental Investigation of the Effect of Program Structure on Program Understanding", *ACM Sigplan Notices*, 12(3), 105-113.

[MACKI91] Mackinlay, J. D., Robertson, G.G., and Card, S. K., (1991), "The Perspective Wall: Detail and Context Smoothly Integrated", In *Proceedings CHI '91 Conference on Human Factors in Computing Systems*, Addison-Wesley, Reading, Massachusetts, 173-179.

[MALHO83] Malhotra, A., Markowitz, H. M., and Pazel, D. P., (1983), "EAS-E: An Integrated Approach to Application Development", *ACM Transactions on Database Systems*, 8(4), 515-542.

[MANEL90] Manel, D. and Havanas, W., (1990), "A Study of the Impact of C++ on Software Maintenance", In *Proceedings IEEE Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, California, 63-69.

[MIARA83] Miara, R. J., Musselman, J. A., Navarro, J. A., and Shneiderman, B., (1983), "Program Indentation and Comprehensibility", *Communications of the ACM*, 26(11), 861-867.

[MORIC85] Moriconi, M. and Hare, D. F., (1985), "PegaSys: A System for Graphical Explanation of Program Designs", *ACM Sigplan Notices*, 20(7), 148-160.

[MORRI81] Morris, J. M. and Schwartz, M. D., (1981), "The Design of a Language-Directed Editor for Block-Structured Languages", *ACM Sigplan Notices*, 16(6), 28-33.

[OBERG92] Oberg, B. and Notkin, D., (1992), "Error Reporting with Graduated Color", *IEEE Software*, 9(6), 33-38.

[OMAN90a] Oman, P. W. and Cook, C. R., (1990), "Typographic Style is More than Cosmetic", *Communications of the ACM*, 33(5), 506-520.

[OMAN90b] Oman, P. W., (1990), "Maintenance Tools", *IEEE Software*, 7(3), 59-65.

[OTTEN84] Ottenstein, K. J. and Ottenstein, L. M., (1984), "The Program Dependence Graph in a Software Development Environment", *ACM Sigplan Notices*, 19(5), 177-184.

[RAYMO88] Raymond, D., (1988), "Preliminary Issues in the Design of a Tagged Text Editor", In *A Potpourri of Prototypes*, Research Report OED-90-01, UW Centre for the New Oxford English Dictionary and Text Research, University of Waterloo, Waterloo, Ontario.

[RAYMO92] Raymond, D., (1992), "Issues in the Design of C++ Browsers", Unpublished Manuscript.

[REISS85] Reiss, S. P., (1985), "PECAN: Program Development Systems that Support Multiple Views", *IEEE Transactions on Software Engineering*, 11(3), 276-285.

[REPST88] Reps, T. and Teitelbaum, T., (1988), *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, London, England.

[ROBER91] Robertson, G. G., Mackinlay, J. D., and Card, S. K., (1991), "Cone Trees: Animated 3D Visualizations of Hierarchical Information", In *Proceedings CHI '91 Conference on Human Factors in Computing Systems*, Addison-Wesley, Reading, Massachusetts, 189-194.

[ROBSO91] Robson, D. J., Bennett, K. H., Cornelius, B. J., and Munro, M., (1991), "Approaches to Program Comprehension", *Journal of Systems and Software*, 14(2), 79-84.

[ROMAN93] Roman, G. And Cox, K. C., (1993), "A Taxonomy of Program Visualization Systems", *IEEE Software*, 26(12), 11-24.

[ROSSA89] Rossak, W., Mittermeir, R. T., and Hochmüller, E., (1989), "Structures for Supporting Software Re-use", In *Proceedings Software Re-Use Workshop*, Springer-Verlag, London, England, 121-130.

[RUBIN90] Rubin, K. S., (1990), "Reuse in Software Engineering: An Object-Oriented Perspective", In *Proceedings COMPCON Spring '90*, IEEE Computer Society Press, Los Alamitos, California, 340-346.

[SAMET90] Sametinger, J., (1990), "A Tool for the Maintenance of C++ Programs", In *Proceedings IEEE Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, California, 54-59.

[SANTA92] Santanu, P., (1992), "SCRUPLE: A Reengineer's Tool for Source Code Search", In *CASCON 1992: Proceedings of the 1992 CAS Conference*, IBM Centre for Advanced Studies, Toronto, Ontario, 329-345.

[SCHAF93] Schaffer, D., Zuo, Z., Bartram, L., Dill, J., Dubs, S., Greenberg, S., and Roseman, M., (1993), "Comparing Fisheye and Full-Zoom Techniques for Navigation of Hierarchically Clustered Networks", In *Proceedings Graphics Interface '93*, Canadian Information Processing Society, Toronto, Ontario, 87-96.

[TAPPR94] Tapp, R. and Kazman, R., (1994), "Determining the Usefulness of Colour and Fonts in a Programming Task", In *Proceedings IEEE Third Workshop on Program Comprehension*, IEEE Computer Society Press, Los Alamitos, California, 154-161.

[VANEK89] Vanek, L. I. and Culp, M. N., (1989), "Static Analysis of Program Source Code using EDSA", In *Proceedings IEEE Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, California, 192-199.

[WEINA89] Weinand, A., Gamma, E., and Marty, R., (1989), "Design and Implementation of ET++, a Seamless Object-Oriented Application Framework", *Structured Programming*, 10(2), 63-87.

[WEISE84] Weiser, M., (1984), "Program Slicing", *IEEE Transactions on Software Engineering*, 10(4), 352-357.