[8] E. W. Mackie, "Waterloo Text Database, System Overview," Technical Report 94–12, University of Waterloo, Computer Science Department, Waterloo, Ontario, March 1994.

[9] ANSI, "American National Standard for Information Systems - Programming Language C, X3J11/90-013," tech. rep., American National Standardization Institute, 1988.

[10] D. R. Raymond, "Lector - An Interactive Formatter for Tagged Text," tech. rep., Univesity of Waterloo, Centre for the New Oxford English Dictionary and Text Research, August 1990.

[11] D. Cowan, E. Mackie, G. Pianosi, and G. d. V. Smit, "Rita – An Editor and User Interface for Manipulating Structured Documents," *Electronic Publishing, Origination, Dissemination and Design*, vol. 4, pp. 125–150, September 1991.

[12] SoftQuad Inc., Toronto, Canada, *SoftQuad Author/Editor User's Manual*, first ed., April 1989.

[13] W. Corporation, *WordPerfect for Windows User's Guide Version 6.0.* WordPerfect Corporation, 1993.

[14] J. Spivey, *The Z Notation: A Reference Manual.* International Series in Computer Science, Hemel Hempstead, Hertfordshire, UK: Prentice Hall, 2nd ed., 1992.

[15] S. Brien, J. Nicholls, *et al.*, "Z Base Standard," ZIP Project Technical Report ZIP/PRG/92/121, SRC Document: 132, Version 1.0, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, November 1992.

guage. Hence, an SGML browser can be used to browse LISP, Ada or COBOL. For languages such as FORTRAN that may be "difficult" to parse, we could replace the DTD-driven parser with a programming-language-dependent parser.

In fact, if the SGML representation of a programming language could be standardized, then every compiler could include an option that would generate the SGML equivalent of the input program. After all, during compilation, all the information about the semantics of the program is collected and it would not be a difficult task to produce an SGML tagged program. For example, as part of the effort to standardize the formal specification language Z [14], the ISO committee decided to include a DTD description of the language with the standards proposal [15].

8 Further Work

Raymond stated "source code can benefit from document formatting and typesetting technology" [6] but it can also benefit from text database technology. SGML is a rich metalanguage able to describe source code. Because many organizations including departments of the U.S. Government are promoting the use of SGML as a standard method for document interchange, there are many different vendors supplying tools to manipulate SGML text.

A software system is defined by a collection of documents which are processed at various stages into human and machine readable formats. Such documentation includes system documentation, user guides, diagrams, program source code, and various graphics. All this documentation could be maintained in an SGML format and supported by various DTDs and style sheets. Hypertext links could be supported by the tag set so that the reader could browse through connected chunks of information.

An important step would be to have the documentation including program code for the entire software system stored in a text database with the proper tools to support browsing, and editing in a WYSIWYG environment. The style sheets could support communication with compilers and similar tools. Thus, the database would support views of the software project and would allow updates through those views. Such a text database might be used to support a software engineering environment, although other issues such as version control have to be considered. We are currently investigating the use of text databases to support software engineering environments with a view to taking advantage of concepts and tools related to tagging and SGML.

9 Conclusions

In this paper we describe the use of SGML to tag program source code for the purposes of visualization and browsing. In particular, we describe how program code written in the language C can be converted to an SGML equivalent, and then how the Waterloo Text Database Browser (an SGML-based tool) can be used to display the code. Using the approach described in the paper we can dramatically enhance the readability of programs by only writing a parser, since many other SGML tools are readily available. Enhancing code for readability and comprehension is just one example of the powerful techniques which can be used in conjunction with tagging concepts. We expect to explore tagging concepts more thoroughly in the context of text databases and software engineering environments.

Acknowledgements

Research described in this paper was supported by IBM Canada, CNPq Brazil and WATFAC.

References

[1] R. M. Baecker and A. Marcus, *Human Factors and Typography for More Readable Programs.* Addison-Wesley, 1990.

[2] A. Goldberg, "Programmer as a Reader," *IEEE Software*, vol. 4, no. 5, pp. 62–70, 1987.

[3] T. W. Reps and T. Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, third ed., 1987.

[4] V. Jacobson, *Unix User's Manual*, ch. tgrind(1). Lawrence Berkeley Laboratory, September 1989.

[5] R. Stallman, *GNU Emacs Manual Ver. 19.* Free Software Foundation, 1993.

[6] D. R. Raymond, "Reading Source Code," in *Proceedings of the 1991 CASCON*, pp. 3–16, October 1991.

[7] C. Goldfarb, *SGML Handbook.* Oxford University Press, 1990. (0-19-853737-9).

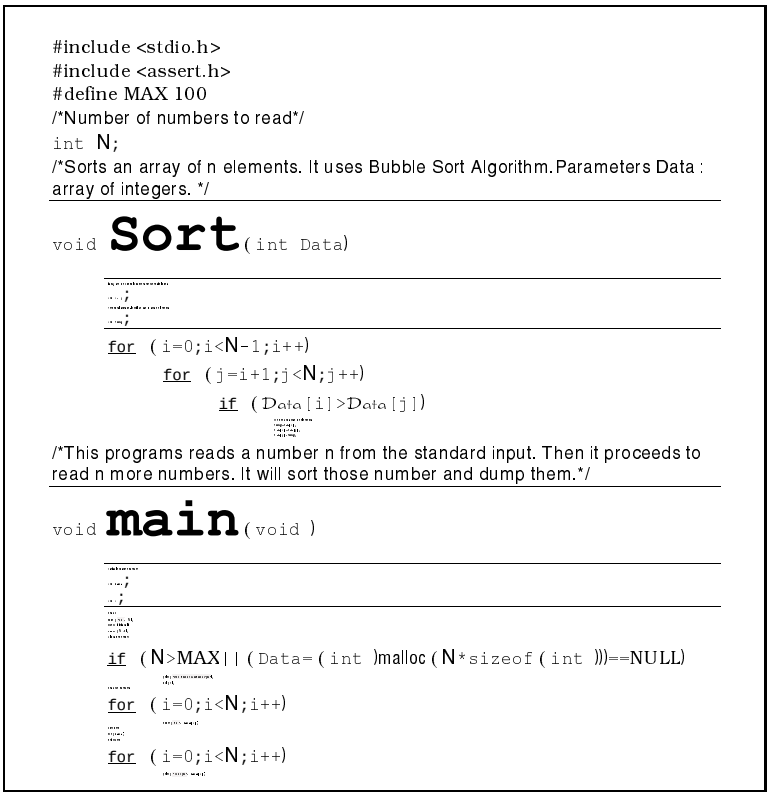

Figure 7: A fish-eye view of a C program

Syntax-directed editors such as Rita [11], the SoftQuad Author/Editor [12] and editors produced by the Cornell Synthesizer [3] could be used to address some of these issues. For example, Rita uses a form of DTD and style sheets to support editing of SGML tagged text. At any point in the document this editor will prompt the user with a list of syntactically acceptable tags. The tags are also separated from the text so the program looks quite "natural". In addition, the style sheet can be used to maintain much of the visualization information which is critical to comprehension of the program text. However, information such as embedded hypertext links would not normally be supported.

Another approach to editing is used by the Waterloo Database Browser. It exports a section of an SGML document to a WordPerfect [13] format, and then formats the output text according to a WordPerfect style-sheet for the document. Then, the user uses the word processor as an editor to modify the SGML document taking full advantage of typesetting for visualization purposes. Once editing is complete, the Browser parses the text and incorporates the changes into the original program file. Of course, this approach means two passes over the text to convert between WordPerfect and SGML formats and maintenance of two style sheets, one for the Browser and one for WordPerfect.

The ideal editor for this task would directly manipulate the SGML file using the DTD and various style sheets to maintain various views for visualization purposes, and also support tags for entities such as hypertext.

7 Using SGML with Other Programming Languages

Different languages have different visualization requirements, but all of them can benefit from appropriate presentation. Since SGML is powerful enough to describe most programming languages, it is only necessary to define a DTD and its corresponding styles and write a converter to produce source code with SGML tags to take advantage of SGML technology. With the exception of the converter between the particular language and its SGML equivalent, all the other tools are independent of the programming lan-

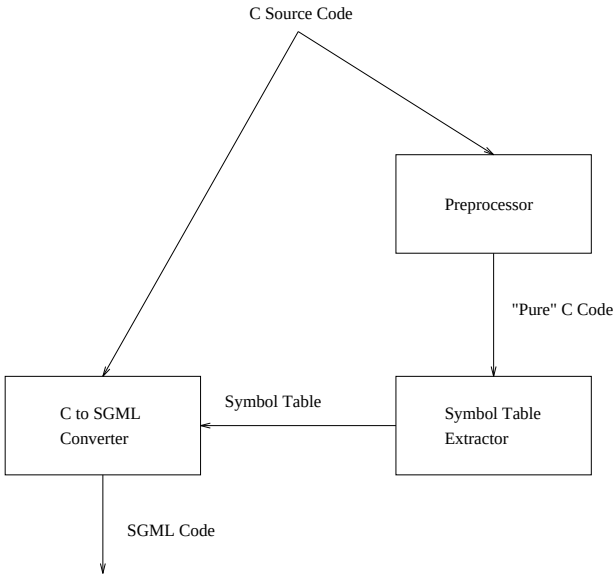

Figure 6: Architecture of the C-to-SGML parser

4.1.3 Browsing the SGML Program

In the last step we define style-sheets for the DTD by declaring properties for each one of its elements. Since the grammar of a text document is modeled as a tree, the properties defined for an element are inherited by all the descendants of a structure node (tag-pair), until such a property is redefined. In addition, properties can be context sensitive; they can vary based on whether an element appears as a child, sibling or descendant of another tag-pair. The visualization properties that can be defined include: font family, font size, indentation, foreground and background color, output before and after the element, line breaks, effects (bold, italics).

Because SGML is standard, different tools can be used to display the generated documents [10]. As better browsers for SGML are developed, better program visualization can be achieved.

The power of visualization using DTDs and style sheets with SGML is further illustrated in Figure 7. In this Figure statements are collapsed, showing a fish-eye view of the program code in which only important control flow statements and definitions are readable.

5 Querying an SGML Program

Some browsers allow the user to navigate through a program, going from usage of an identifier to its definition. But when more complicated searches arise, programmers rely on simple search tools such as grep. However, such searches lack information about the structure of a program; they view a program as a single stream of characters divided into lines.

Searching can benefit from the SGML representation of source code. As was mentioned previously, hypertext capabilities support linking an identifier to its definition. This is not restricted to function names, but also type definitions, parameters, variables and defines statements. In addition, some SGML tools are able to do context-dependent searches such as locating a given string inside a given structure. For example, usage of a variable i in the control expressions of while statements; or looking for all the definitions of the variable *foo* inside a block.

6 Editing SGML Tagged Programs

As is evident from Figure 3, ASCII text with embedded SGML is hard to read, and thus editing will be a cumbersome, if not unpleasant task. Remembering the specific tag set for segments of program text[4], and placing the tags to ensure a syntactically correct document will also create problems. In addition, if we edit the tagged document directly, then all the visualization information that can be generated from the tags will not be present.

[4]There are a large number of tags for C and remembering them may represent quite a difficult task.

```
<style id="fundef">
    <element>            fundef
    <type>               text
    <block>
        <break>          before
        <border-pos>     top
        <border-line>    single
        <border-colour>  cyan
</style>

<style id="func-var">
    <element>            func-var
    <parents>            linkpt
    <ancestors>          fundef
    <type>               text
    <font>
        <size>           +24 pt
        <bold>           on
</style>

<style id="declor">
    <element>            declor
    <type>               text
    <block>
        <left-indent>    0.5 in
</style>

<style id="func-var-call">
    <element>            func-var
    <parents>            link
    <type>               text
    <font>
      <name> SwitzerlandCondensed </name>
      <fore-colour>    black
</style>
```

Figure 5: A simple partial style-sheet

macros. Because these two constructs can appear almost anywhere in the program text, the DTD would be quite complex. For this reason some assumptions were made about the structure of C programs. We decided that macros can appear anywhere an identifier can, and that comments can appear between statements and next to identifiers. If a comment appears in an "illegal" place it should and can be moved to the closest legal location.

Another important issue is the level of detail desired in the program presentation. Currently, SGML tools have restrictions in the nesting levels that they support, and parsing trees for C programs exceed the limits of most SGML tools. Furthermore, not all the productions in the C grammar are relevant for visualization purposes. It is an interesting problem to experiment with different DTDs that describe C programs at different levels of detail, and try to find the smallest DTD that includes the information necessary to achieve appropriate program visualization. Since C expressions generate very deep parsing trees, we decided not to include all the productions of C expressions in the SGML DTD and instead, caused expressions to generate a sequence of operators, constants, and identifiers. Even with this simplification, it is still possible to identify semantic information inside expressions. For example, overloaded operators such as the asterisk (*) can be tagged for uses such as a pointer reference and a multiplication operator. As a consequence, the two different uses of this operator can be presented to the reader in different formats.

4.1.2 An Automatic Converter from C to SGML

Once a DTD was defined, the next step was to write a parser that would convert C code to its SGML equivalent. Figure 6 is a block diagram of the parser.

The source code is first preprocessed by a standard compiler. The reason for this is to take advantage of the C preprocessor to evaluate the preprocessor instructions in the source code. The output code will be "pure" C code. This code is parsed and its symbol table is extracted. The symbol table will contain information about all the identifiers defined in the program and the location of their definitions.

This information along with the original source code is used as input by a second parser that generates the SGML output. The reason to do the conversion with two different parsers was simplicity. However, it is possible to write a C-SGML converter that performs a single pass on the source code, but such an approach would require a more complex parser.

```
<!ELEMENT FunDef          - -  (DeclSpec?,declor,FunBody)>
<!ELEMENT DeclSpec        - - ((StoClSpe|TypeSpec)?)>
<!ELEMENT TypeSpec        - - ( CHAR | SHORT | INT | LONG | SIGNED |
                                UNSIGNED | FLOAT | DOUBLE | CONST |
                                VOLATILE | VOID | stunspec | enumspec |
                                TYPENAME)>
<!ELEMENT declor          - - ( pointer?,DirDecl)>
<!ELEMENT DirDecl         - - ( Ident
                              | (LPAREN,declor,RPAREN)
                              | (DirDecl,((LBRAC,ConsExpr?,RBRAC)|
                                (LPAREN,(ParTyLis|ParIdeLi)?,RPAREN))))>
<!ELEMENT Ident           - - ( linkpt|link|#PCDATA)>
<!ELEMENT  (linkpt,link) - - ((def-var|glob-var|func-var|loc-var|parm-var)?,
                              #PCDATA)>
```

Figure 4: A simple partial DTD

tended to store, browse and query text databases, using SGML as the underlying markup language. Each document stored in the database can have a different DTD as well as any number of corresponding style-sheets. The Browser supports hypertext links. Those links are part of the DTD and therefore defined using SGML.

4 Using SGML to Display Source Code

Since most programming languages are context free, SGML can be used to describe the structure of valid programs in any of these languages[3]. Figure 3 shows a small section of the code from Figure 1 with embedded SGML tags. The conversion between a program in a context-free programming language and a fully tagged program such as the one in Figure 3 can be performed automatically using standard parsing methods. To describe SGML documents containing programs from a given programming language, it is necessary to:

1. Define a DTD that describes the syntax of the language. If the language is described in BNF it is easily converted into an SGML DTD.
2. Automate the conversion of programs to their SGML equivalent. This task can be performed with a parser for the given programming language. For example, a program can be converted into a parse tree and then the SGML version can be generated.
3. Define various style-sheets to map DTD elements to various formats.

[3] Some languages might be ambiguous, but a valid program in such a language has a corresponding parse tree and such a parse tree can be represented in SGML. Ambiguous languages make compilation difficult but the corresponding document can still be represented in SGML.

4.1 Converting C to SGML

At each stage in the conversion to tagged text a number of specific issues were encountered that needed to be addressed. These issues are discussed in the next subsections.

4.1.1 Constructing the DTD

It is possible to convert the entire ANSI standard grammar [9] for C to an SGML DTD but some problems are encountered. Some lexical elements of C programs such as comments are not included in the grammar for the language; they are removed by the lexical analyzer before the program is compiled. Preprocessing commands are also usually evaluated and removed before compilation. Therefore, the grammar for preprocessing is separated from the grammar for the actual language. In order to be effective, the program tagged with SGML has to reflect the way the programmer wrote the code, and not the code just before the compilation step.

SGML is capable of describing a grammar for C that contains both comments and preprocessing

3 The Descriptive Markup Language SGML

Plain text relies on formatting to include information about the text's structure. For example, the section title of an article is usually printed in boldface with a larger font, and the first line of each paragraph is indented and separated from the previous paragraph by a blank line. This structural information is not included in the text, but presented through its appearance.

```
<FunDef><DeclSpec><TypeSpec><VOID>
</TypeSpec></DeclSpec><declor><DirDecl>
<Ident><linkpt id=main><func-var>
main</func-var></linkpt></Ident>
</DirDecl>
```

Figure 3: Section of code from Figure 1 with embedded SGML tags

Descriptive markup techniques use a different approach; special markup tags are embedded in a text document to add information about the document's structure. For instance, the descriptive markup approach might use <paragraph> to indicate where a paragraph starts and </paragraph> to indicate where it finishes. Figure 3 shows a small section of the code from Figure 1 augmented with embedded tags from the markup language SGML. The tags can be distinguished from the text because they are always surrounded by angle brackets, and so it is quite easy to recover the original program. The usual convention in SGML has tags with the form <id> to identify the start of an element, and tags of the form </id> for the corresponding end. Descriptive markup is used to define the structure of a document rather than the document's appearance, although we will show later that the two concepts can be related.

The Standard Generalized Markup Language (SGML) [7], an ISO standard (ISO 8879), is becoming widespread as a method of incorporating descriptive markup tags in text documents. The basic concept behind descriptive markup languages is that the structure of any document can be described with a context-free grammar and that documents fall into structural equivalence classes such as contracts, memoranda, business letters and books. SGML provides a metalanguage for defining tag sets to mark each of these different types of structures within a document. When a document is created these tags are embedded in the document and usually bracket the corresponding structure. For example, the function variable `main` in Figure 3 is bracketed by the tags <func-var> and </func-var> which indicate the start and end of this structure. A Document Type Definition (DTD) or grammar for each class of document can be defined and used with a document parser to verify that a tagged document complies with its definition class. In other words, any valid SGML document should have a corresponding DTD.

A partial DTD for the simple tag set used in Figure 3 is illustrated in Figure 4. This class grammar definition is similar to the definition one might encounter for the syntax of a programming language. For example, the first line of the DTD states that there is a function definition (`FunDef`) and that it encloses an optional declaration specification (`DeclSpec?`), followed by a set of declarations (`declor`), and then by a function body (`FunBody`). The remainder of the specification segment defines some of these constructs. Each of these constructs would be bracketed by a set of tags that are generated from the name of the structure. For instance, the structural element `FunDef` would generate the tags <FunDef> and </FunDef>.

An SGML browser can be created that will take as input a DTD, an SGML document compliant with this DTD, and a style-sheet for the DTD. The result is the SGML document typeset according to the style-sheet specification. This approach gives the user the flexibility to define different style-sheets for any particular DTD and as a result, the same documents can be displayed in many different formats, without having to be modified. A style sheet for the DTD in Figure 4 is shown in Figure 5. Most of the information in the style sheet is self-explanatory. The <element> names the tag pair and the remaining information describes the manner in which the document will be displayed. If a dynamic medium is used for display it is possible to define hyperlinks; this technique allows structures to be connected to other related information. Note that style is context sensitive and can depend on the parent or some other ancestor of a structure. Thus, there can be several style definitions for the same tag pair.

SGML is growing in popularity because of a number of document portability and standardization initiatives. This popularity has prompted the development of a large number of tools for processing and viewing SGML documents. The prototype we have developed for browsing and viewing program code uses the Waterloo Text Database Browser [8], a tool in-

```
/* This program reads a number n from the standard input. Then it proceeds to read n more
numbers. It will sort those numbers and dump them.*/

void main(void)
{
/* Variable to store number */
int *Data;
int i;
/* Read N */
scanf("%d",&N);
/* Check N is valid */
assert(N > 0);
/* Allocate memory */
if (N > MAX || (Data=(int*)malloc(N*sizeof(int)))==NULL){
   printf("NOt enough memory\n");
   exit(1);
} ...
```

Figure 1: Plain source code

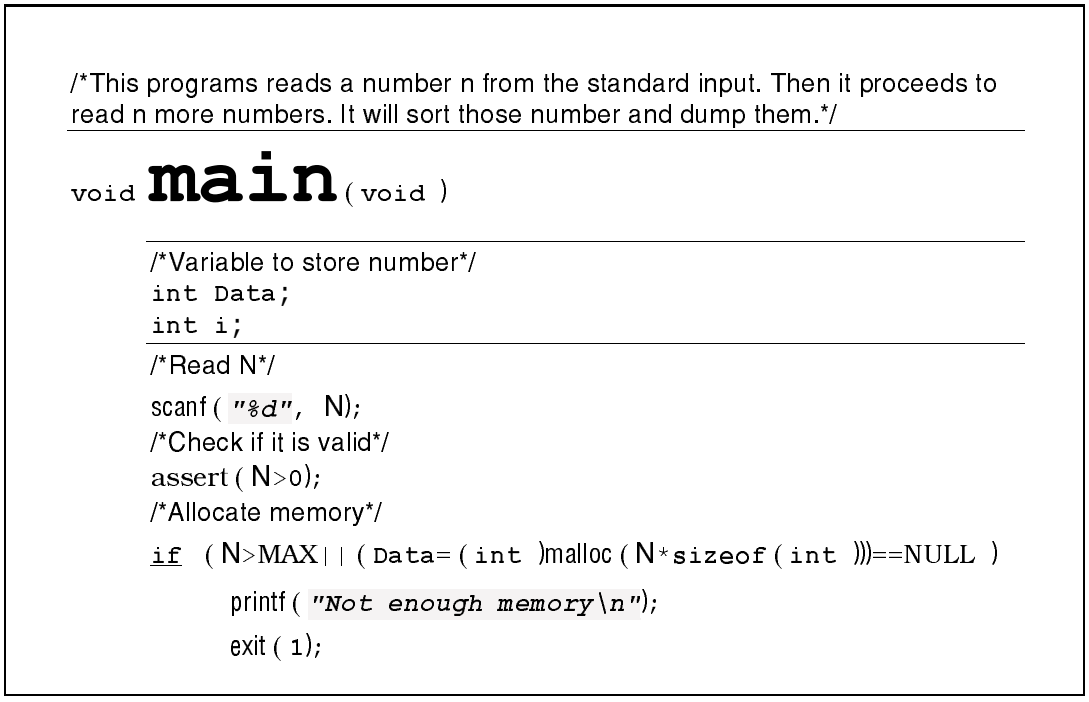

Figure 2: Using semantic and syntactic information to visualize code

as `indent` which indents C code based on user specifications, have been developed to address the problem of spacing and indentation. Syntax-directed editors [3] have been created that support readability and assist the programmer by enforcing syntactically correct program entry and modification. Work in this area has also focused on enhancing lexical information. Examples of tools in this area are pretty printers such as `tgrind` [4], that automatically indents the code to a given style and prints reserved words in boldface, and Emacs [5] a program editor that can display each lexical element of a program with different color or attributes (bold, underscored, italicized). Other tools allow the programmer to browse through source code giving the programmer a "pretty-printer" view of the code plus hypertext capabilities: the user can point to the usage of a function and jump to its definition

Baecker and Marcus proposed that programming languages must include a description of how they must be typeset as part of their definition [1]. They also emphasized that program visualization must benefit not only from lexical but from syntactic and semantic information. In particular they focused on C, presenting different ways in which the semantic information of a C program can be used to enhance readability.

Source code presentation can also benefit from recent research in text databases. Different views of the program text can be used to highlight various aspects of its structure and semantics. For example, fish-eye or outline views showing only the specification of function and procedures might be useful. Browsers of text databases could take advantage of embedded navigational information such as automatically generated indexes, and hypertext links. Raymond [6] proposed to use text database technology to enhance readability of source code. He demonstrated that text browsers can be successfully used to traverse and read source code. However, his model only exploited lexical information.

We propose that the presentation of source code can benefit from both areas: code visualization and text databases. Using descriptive markup of the source code (which describes the structure of a document rather than its formatting) the semantics of the program (equivalent to a parse tree) can be easily embedded in the code, and then stored in a text database. The text database viewer can be used to typeset the program as Baecker and Marcus proposed. In addition, the navigational and query capabilities of the text database can be used to browse the source code. The database could support different views of the program code highlighting features such as call structures, outlines of program structures, local pseudo-code or assertions.

As a result, locating and reading code will be an easier task, programmers will take less time to become familiar with the source code, and the time to maintain a given system should be reduced. In this paper we examine techniques for embedding descriptive markup tags in a program text and show how these can be used to enhance the typography and readability of the program. We briefly examine text database research and describe some directions in this area that could be used to heighten comprehension of programs.

2 Typesetting Source Code

The fundamental idea behind code visualization is that almost every single grammatical element of a program carries potential information that can be used to enhance its presentation.

Figure 1 shows a small section of source code[2], written in C. Figure 2 shows the same code presented through a text database browser. Although both sections of code are identical and perform exactly the same function, they are presented to the viewer in radically different formats.

Notice in Figure 2, that the presentation of each token of the source code is intended to convey information about the token's characteristics. Normally, readers of the program have to know either about the context in which a section of code is embedded or spend some time analyzing it. For example, before readers can determine whether the usage of an identifier is a function name or a global variable, they have to locate its definition. With proper typesetting, the type of an identifier can be quickly identified.

In particular, in the previous example, the name of the function `main` is presented in its definition with a large font. The area of definition of variables is clearly separated with two lines. Local variables, global variables, function names, preprocessor definitions, and comments all use different fonts.

In this case, the hard-copy is a static presentation of the source code. An interactive view of the same code will show hypertext links from the usage of an identifier to its definition, so the readers can easily check the relationship between the two entities.

[2] We do not claim that the programmer was careful in the organization of the program for presentation.

Enhancing Code for Readability and Comprehension Using SGML

D.D. Cowan D.M. Germán
Department of Computer Science
University of Waterloo
Waterloo Ont, N2L 3G1

C.J.P. Lucena A. von Staa*
Departamento de Informática
Pontifícia Universidade Católica
Rio de Janeiro, Brazil 22453-900

Abstract

Reading and understanding programs is a key activity in software reengineering, development, and maintenance. The ability of people to understand programs is directly related to the ease with which the source code and documentation can be read. Thus, enhancements to the style of presentation should heighten this comprehensibility. We describe methods that use markup languages such as SGML to embed information about the syntax and semantics of a program in the program code, and then show how these can be used to enhance its presentation style. We also briefly discuss the extension of these markup language concepts to text databases, and indicate how they can support various structural views of the code through browsing techniques associated with database queries.

1 Introduction

A key step in reengineering a software system is extracting the enterprise rules and meaning in order to re-implement the process using alternative software technologies. Reengineering requires a careful scanning of the documentation and code in order to develop the necessary understanding of the software system. Thus, any process that enhances readability in support of comprehension is a positive step in the reengineering process. Of course, enhanced readability is useful even for software maintenance and development; in fact, for any software process that requires an understanding of the high-level semantics of a system.

Although readability depends upon the programming language and on the person reading the code, and is difficult to define exactly, there are some general rules that can be applied to make code more readable. Code should be in a format that shows relationships of subordination; for example, the body of a procedure should be indented from its declaration, or statements under a control statement should be grouped and indented. Names should express the semantics of the concept they represent. Appropriate comments that express the high-level semantics of a program and perhaps contain formal specifications should be included. Some languages are easier to read than others, and in some cases, readability has been an issue when designing some languages.

Visual enhancement can also be of assistance when understanding source code. Program visualization focuses on enhancing presentation of existing programs, their code and documentation. According to Baecker [1] the goal of program visualization is to facilitate the clear and correct expression of the mental images of the writer of computer programs and the communication of these mental images to the readers of such programs.

Despite the importance of readability, most programs are not readable [2]. This lack of readability is caused by a number of factors. Programmers do not wish to take the time to make their code readable since it uses "valuable" production time, and programs in most programming languages are inherently difficult to read. In addition, existing tools are not widely known or available[1] in many sectors of the software development community.

While other forms of textual information have greatly benefited from the progress in typesetting and bit-mapped color monitors, programmers have barely taken advantage of many of these advances. However, the programmer can and should play an important role in making code readable, and should be supported by editors and visualization tools that enhance readability.

A number of tools have been designed to enhance readability through visualization. Tools such

*C.J.P. Lucena and A. von Staa are Visiting Professors in the Computer Science Department at the University of Waterloo

[1] Several tools are available under operating systems which until recently were not commonly used for most software development.