NASA-CR-192798

JPL Publication 92-12

Conjunctive Programming:

An Interactive Approach to Software System Synthesis

Robert C. Tausworthe

August 1, 1992

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL Publication 92-12

Conjunctive Programming:

An Interactive Approach to Software System Synthesis

Robert C. Tausworthe

August 1, 1992

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Abstract

This report introduces a technique of software documentation called conjunctive programming and discusses its role in the development and maintenance of software systems. The report also describes the *ConJoin* tool, an adjunct to assist practitioners. Aimed at supporting software reuse while conforming with conventional development practices, conjunctive programming is defined as the extraction, integration, and embellishment of pertinent information obtained directly from an existing database of software artifacts, such as specifications, source code, configuration data, link-edit scripts, utility files, and other relevant information, into a product that achieves desired levels of detail, content, and production quality. Conjunctive programs typically include automatically generated tables of contents, indexes, cross references, bibliographic citations, tables, and figures (including graphics and illustrations). This report presents an example of conjunctive programming by documenting the use and implementation of the *ConJoin* program.

Acknowledgements

The author would like to acknowledge and thank Brian Beckman, Thomas Fouser, and Karen Owens for their encouragement, review, and suggestions in the writing of this report, and Ellen Reinig and Barbara Amago for their expertise in editing and preparing the material for publication.

Contents

Appendices

List of Figures

1 INTRODUCTION

1.1 Documentation Problems

One of the traditional problems with maintaining and reusing computer programs is understanding—even by the authors, after a period of time has elapsed. Another is consistency among the software artifacts (program and library code, documents, linkage-edit files, databases, etc.) as time progresses and as adaptations evolve within multiple platforms.

Each programmer has a particular individual manner of writing programs that includes organization and composition, choice of algorithms, method of indentation, density of comments, naming of variables, extent of information hiding, application of modular packaging, and form of expression. In addition, the visual appearance, including typography and graphics, may vary according to the programmer's personal style. It remains a fact that a program written by someone else may be very difficult to understand, even when produced by acceptable development practices.

Some advocate "self-documenting code," or intensely annotated programs that contain all the information that the programmers believe is necessary. Making a program entirely readable by itself is, in some sense, analogous to making a circuit board or computer chip layout readable by itself: It is a difficult process usually producing unsatisfactory results that are not appropriate for the medium. Rather, readability gains where explanation, clarity, structure, visualization, layout, use of color, and rendering can be separately and adequately treated.

In the past, software artifacts were viewed by humans as documents. In the automated world of today, however, the concept of what a document is has undergone a vast change. The computer and communications industries have made enormous progress in giving humans immediate access to huge stores of information. The majority of this information, captured in printed material, libraries, and computer files, is ultimately for human perception. Some information is fully formatted for immediate and direct perception as printed matter. Some is fully formatted, ready to be rendered into perceivable form (e.g., PostScript files, digital video, and sound recordings). The rest is stored unformatted, but in such a way that it is possible to format, process, or otherwise render it for direct perception (e.g., in databases and in multimedia hypertext documents). In the future, expansion of the documentation concept to include an even wider context of perception will, no doubt lead to new needs for people to locate, perceive, and interactively understand not only software artifacts, but large bases of information of arbitrary types as well.

Software documentation problems have sprung from a number of diverse sources, among which are the lack of accepted, effective industry-wide standards; the conspicuous absence of software documentation courses in university curricula; the inadequate training within corporate development programs; the emphasis within existing standards on form, rather than on substance and sufficiency; the uncertainty in, ignorance of, and insensitivity of developers to the needs of readers; the inability to provide the proper level and content of material needed by users; the prohibitively high cost of providing highly readable, textbook-quality documentation; the pressure by managers and customers to "get the code working"; the deficient allocations of resources for providing proper documentation; the shortfall of practical methods and friendly, integrated tools for doing a good documentation job; and the lack of cost-effective means for combating the continuing entropic divergence in consistency among artifacts.

In a recent visit to the library of the JPL Software Resource Center, an organization formed to guide the improvement of the Laboratory's software engineering methods and practices, I found only one text completely devoted to documenting a program [1], and one other on diagrammatic methods applied to programming [3]. The LaTeX [4] reference manual was there, but that text tells how to compile a document, not how to document a program. Some of the library's texts devoted space to defining the documentation problem, justifying why projects should improve their documentation and giving examples of document outlines, code annotation, review guidelines, and subjective acceptance criteria. However, not one of the texts offered a real, codified, practical and comprehensive approach to program understanding. Such works may exist, but at JPL their teachings are not part of the culture.

Most of us in the software industry have taken high school and university courses in composition. We have at least been exposed to the art of literary discourse, expression, style, and organization. If we have forgotten, or otherwise need to (re)learn how to communicate in writing, there are books, reference works, and computerized tools readily available to help us at all levels of ability. It would therefore seem more natural for us to be able to explain something in our natural language than in a syntactically more restricted, awkward computer language form. But a formula for producing software understandability is a tall order. When applied by the practitioner, it must be capable of causing objects to be created that will communicate the intended information to the intended readers of presupposed intelligence levels, areas of skill, and cognitive styles [2]. This formula may not exist.

In my experience, the best software documentation exists in journal articles and in textbooks on algorithms. In each of these understandability has been scrutinized by referees and editors. Unreadable works are rejected before their release to the public, and works of inferior quality that leak through the review process are generally short-lived. Those that survive serve as examples of the high quality that can be attained.

In one not atypical work [5], I noted that the ratio of space devoted to accompanying narrative, mathematical formulae, derivation, figures, tables, and other explanatory material exceeded the space devoted to displaying the code by a factor ranging from about two to ten! If this one sample is anywhere near to being a valid indicator, and if textbook quality is what is really required for reader understanding, then industry must expect its software projects to require a far greater proportion of time and resources on documentation than any other development pursuit. Indeed, most projects of any size will not be able to afford textbook-quality documentation.

Sadly, I do not have a general approach to solving the overall documentation problem either. Far more resources than I have, and wiser faculties than mine have attacked the problem vigorously for many years, with only meager success. This report does, however, offer a concept, a practical approach, and a simple tool to demonstrate a practice that, I hope, will help to increase programmer sensitivity to the needs of readers, assist in producing good documents in *normal documentation time*, permit separate development and documentation efforts to take place, and promote consistency among software artifacts. I believe that any level of content a project may decide upon for a document, from copious detail in meticulous, multi-linked, multicolored hypermedia, to no documentation at all, can be accommodated within this concept. Whatever form of documentation is chosen, from textbook or journal rigor, to Department of Defense standards, to undisciplined personal quirks, can be created. As long as the sources of software artifacts are accessible within the system environment, the documentation process should be able to apply the technique presented in this report.

1.2 Document Markup

One concept central to the approach of this report is that ancillary information may be benignly inserted into software artifacts so as either to enhance the primary information content or extend the utility of those artifacts. *Markup* is a type of information that is introduced into data to convey special interpretations. It is so called because of its resemblance to the markings that editors make in drafts of paper documentation. Markup makes use of "start tags" and "end tags" that respectively precede and follow each logical portion of the data. Tags are specially formed so that the markup can be recognized and processed separately from data that surround it.

Markup tags fall into four categories:

(1) Descriptive markup, which defines the structure and appearance of a document. These tags identify such items as sections, subsections, citations, references, fonts, and so on.

(2) Entity references, which are requests for objects to be moved into the document at (or near) the point of reference. The objects themselves may reside elsewhere within the medium containing the reference, or externally.

(3) Markup declarations, which are statements that control how the markup is interpreted. These can be used to define objects directly and also to

create additional markup descriptions.

(4) Processing instructions, which are instructions to the processing system, in its own language, to take specific actions. Unlike the three other types of markup, this markup is system-dependent, and perhaps application-dependent, as well.

Markup tags do not need to be physically embedded within the information medium itself. They can be maintained in a separate database that defines the type, location, and other particulars of each mark.

American industry and the international community have developed the Standard Generalized Markup Language (SGML) [6], and are in the draft stage of developing HyTime, a Hypermedia/Time-based document-structuring language [7] built on SGML. These standards offer ways to make "information about information" interoperable. An introductory article on HyTime appears in [8].

In a markup system, when a document type (such as a book, an article, or a report) is defined, a distinction is made between the information to be presented and the instructions for rendering that information for perception. This is a fairly straightforward process for many traditional types of documents, where information consists of printable words, punctuation, and simple graphics. Formatting a document follows a style guide, or specification that associates rendering instructions with generic markup tags. All that is necessary to reformat an entire document is to apply a different style guide.

Markup tagging offers another, even more significant, benefit: collections of tagged objects may be queried like databases. By using special tags for interlinking objects, information products can access the contents of other information products by reference. The use of automated query data involving actual objects, rather than manually generated surrogates of those data, can help immunize the produced documents against obsolescence. Moreover, the information in tagged objects remains available for other uses totally unforeseen when the objects and tags were created.

Markup tagging for document rendering and style must be recognized by the target document-processing system. Tagging for location of information must be recognized by the agents that establish and utilize linkages among objects. The placement and maintenance of linkage tags are the responsibilities of authors and owners of the information objects queried.

1.3 Literate Programming

Donald E. Knuth of Stanford University released a markup style utility in 1983 named **WEB** [11] as part of a project that also developed the TEX markup system and document compiler [12]. The use of **WEB** by Knuth [13] and others [14, 15, 16, 17], led to the *literate programming* paradigm, a concept in which programs are treated as works of literature. The idea behind this approach to programming is that software (code, documentation, and associated data) can be made so readable and interesting that it may actually be read for entertainment. If an author has craft and style, and if a program does something interesting, or does something in an interesting way, then the presentation to the reader can be enjoyable, as well as enlightening.

Literate programs have an orientation and expression that differs from other programs. The literate program is a special mixture of expository text that tells the reader what the program does, how it does it, and why it does it that way. Not by accident, it also tells the compiler how to build the program. Thus, the program's documentation, code, and data are integrated and in one place, possibly in a single computer file. They are created and maintained together as a unit.

Unlike any other procedural programming language, **WEB** programmers are not required to present the pieces of a program in any specific order. Rather, they can be arranged in a natural order, which Knuth refers to as *stream of consciousness*, and they can be organized into segments that accommodate the needs of structured programming. Rather than developing "self-documenting code," literate programmers create "self-coding documents."

WEB comprises two program utilities, **WEAVE** and **TANGLE**, which process the literate source files and create, respectively, a document file to be processed by TEX and a code file to be processed by a Pascal (or other) compiler. The details of **WEB** and literate programming are described in Wayne Sewell's well-written book, *Weaving a Program: Literate Programming in WEB* [18].

2 CONJUNCTIVE PROGRAMMING

This section describes an alternate to literate programming, a concept that I call conjunctive programming.[1] Because it is simple, easy to learn, flexible, and compatible with almost any documentation regimen, many may find it to be an appealing and useful adjunct to literate programming.

The idea is not new. Like modularity and object-oriented methodology, many may realize they have been doing something like it for years without knowing that was what it was. While this report primarily treats generation of conventional text-and-graphics types of documents, extensions to multimedia and hypertext forms will be readily recognized. A hypermedia example of a similar concept, called `Intermedia` appears in [10].

2.1 The Concept

In conjunctive programming, an object is any form of electronically accessible information. A software system is comprised of the set of objects that define, build, configure, and operate the system. These objects, therefore, contain much (but rarely all) of the information needed to understand the system. This information is current and accurate, because it is the real McCoy, not an artificial surrogate maintained separately. It is a resource that can be used and reused by excerpting appropriate portions into desired external documented forms.

The verb **conjoin** means [19] to unify and integrate separate entities together for a common purpose. The adjective form **conjunctive** means connected, conjoined, and composed of, or functioning as, a combination. **Conjunctive programming**, then, is the assembling together of information excerpted from real software artifacts, such as programs, items in accessible libraries, version-specific configuration data, link-edit code, scripts, other documents, outputs from other processors (e.g., structure analyzers, pseudocode systems, statistical packages, etc.), and all other information required to create a document with designated levels of detail, content, and production quality.

Producing a conjunctive program is somewhat similar to the "cut-and-paste" process one exercises when writing a document using a text editor or word processor on various sources of online data. It differs in that after an excerpt to be reused has been located, it is not pasted directly into the document at that time. Rather, a *window* to this excerpt is pasted into the conjunctive document instead. The window is alive; if the excerpt changes, the view of it in the window changes also. When the time to produce the target document finally arrives, a software tool scans the source document, taking snapshots through all the windows, replicating their current views.

The product of this process may be likened to a collage, in that it is an artistic composition made of various diverse fragments of heterogeneous materials

[1] Warning: *Conjunctive literitis* is a mental condition thought to be linked to the overzealous devotion to this practice.

assembled (glued) together. However, this analogy cannot be further extended to describe the conjunctive programming process, for there is no listed adjective form describing the collage-making activity. Coined terms such as "collagenous," "collagenative," and "collageneric," seem awkward, contrived, and inadequate. It is true that "collage" can be used as a transitive verb, but it sounds awkward to describe conjunctive programming as a process of "collaging" bits and pieces of existing information together.

Conjunctive programming is a generic concept, specific instances of which are defined by the environment of automated utilities that provide for acquiring, editing, interrelating, integrating, navigating, and finally rendering the information products. In concept, conjunctive programming does not require markup tagging for these functions, but as a practical matter, it offers a great convenience. Intermedia is an example of a highly integrated conjunctive programming environment in which the actual linkage mechanisms are largely hidden from user view. The degree to which conjunctive mechanisms are invisible within a programming environment that capitalizes on the benefits they provide attests the level of sophistication and utility of the environment.

But the use of conjunctive programming does not have to be restricted only to highly integrated, grandiose environments. Rather, as illustrated in this report, it can be applied productively even in less sophisticated, relatively heterogeneous environments. Processors like TEX and its dialects provide a significant portion of the mechanical advantage needed, permitting people with ordinary text editors to produce high-quality, typeset documents. LATEX and AMS-TEX are powerful style adjuncts of TEX that help create and render titles, abstracts, tables of contents, numbered sections, footnotes, tables, figures, indexes, cross-references, bibliographies, and citations. With TEX and LATEX serving as its basis, an effective conjunctive programming environment additionally only requires a simple capability for recognizing and transcribing entity references among software artifacts. Existing development tools, such as file analyzers, type font managers, special formatters, program tree plotters, cross-reference generators, spelling checkers, thesauruses, and **MAKE**[2] programs with dependency files, can also augment the environment.

Conjunctive programming in TEX can produce the same superior document quality as produced by literate programming in **WEB**. A TEX-based conjunctive programming environment and information flow are illustrated in Figure 1.

The conjunctive programming approach to software development is differently oriented than that of literate programming, even though one hopes that conjunctive programs can be equally literary in their final products. Conjunctive programs, like literate programs, are displayed as interspersed narrative, code, and data, printed in typeset quality. Actual source code, configuration data, link and library directives, **MAKE** files, and the like, are all sewn together

[2]**MAKE** is a utility (from UNIX) for automated regeneration of artifacts that are dependent on other artifacts, which may change from time to time.

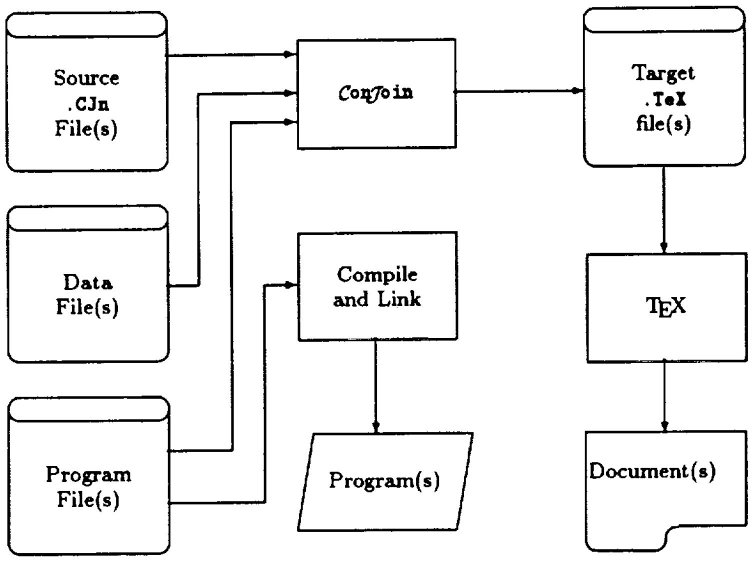

Figure 1: TeX-based Conjunctive Programming Information Flow.

into a format not unlike that of literate programs. The difference is that conjunctive programming presumes that the code, data, documentation, and other artifacts are maintained in separate, conventional forms that are then merged into the integrated final form seen by the reader.

The practice of conjunctive programming may be pursued at different levels: art, craft, discipline, or mere application. The *art* of conjunctive programming exhibits conspicuous ingenuity and creative imagination, as may be manifested by an engaging manner of description, innovative articulation, clever insights and revelations, and curious mysteries for the reader to ponder and solve. *Craft* is evidenced by technical accuracy, rigor, and expert workmanship. Conjunctive programming *discipline* is the observance of orderly, sound, and systematic processes which conform to recognized sets of rules, standards, or guidelines. The mere practice of conjunctive programming does not, in itself, attest the level of talent and skill that is at work.

Conjunctive programs contain linkages into real-world objects that may be subject to a form of entropic degradation caused by evolutionary changes in the system. As a practical matter, it is impossible either to prevent or counter this degradation in a timely fashion without the cooperation of the individuals responsible for each linked-to object, or without elaborate automated linkage management, or both. Both require the use of markup standards, automatic recognition of changes, and establishment of communications and coordinated actions among the individuals and artifacts involved. The integrated **Intermedia** system for maintenance of linkage information in hypermedia documentation is discussed in [10].

Documents may suffer from incompleteness, inconsistency, noncurrency, and asynchrony, if improperly written or maintained. Whether a program description is complete is perhaps more subjective in conjunctive programs than in literate programs, because only the code segments deemed of interest may have been abstracted into a conjunctive document, whereas the entire program must appear in the literate program. On the other hand, conjunctive programs can also be complete; they can also produce additional reports and papers about selected algorithms, data structures, and the more essential aspects of the program. Thus, conjunctive programming can either be equally elaborate as, or considerably simpler than, conventional literate programming, at the whim (and according to the resources) of the author.

Conjunctive programming allows freedom of expression for whatever good it may inspire. It also serves those who must more rigorously conform to industry or government standards. Certainly the tools of conjunctive programming are capable of capturing and expressing far more literate examples of programming than will be seen here. The fault lies not in the conjunctive programming concept, nor with the use of the tools, but rather, as in all programming, in the capability of the practitioner and the time resources available.

Consistency is the degree to which artifacts are internally and externally free from contradictions. Consistency conveys compatibility in content, style, and terminology. One form of consistency relates to currency, which is the degree to which the documentation correctly applies to the existing system. Literate programs are guaranteed to be current in coding specifications by their construction. Parts of a literate program may be incorrect or in conflict, but these parts always define the program totally, for better or for worse. Conjunctive programs draw system information directly from the system database. Whether copied completely or correctly, the objects conjoined are true reflections of what is in the system. Automated updates of documentation using **MAKE** whenever system changes are made will help to keep conjunctive programs current. The *ConJoin* tool currently does not search for inconsistencies but will complain whenever directives are inconsistent with the content of the artifact being accessed.

Synchrony is the degree of consistency between referenced portions of system files and the contexts of the documents in which these excerpts appear. If maintenance deletes a code segment previously excerpted into a document, asynchrony appears as a "hole," or error, in the document context. If a program modification erroneously moves an entity reference key, the code viewed may not pertain to the document subject matter at that point. Synchronism of document text and excerpted material requires robustness in the entity reference scheme and immunity to changes in the system. More about the creation of robust entity reference keys appears later in this report, in Section 5.2.

2.2 Life Cycle Considerations

Conjunctive programming methodology need not be limited to the code development and maintenance portions of the software life cycle. There is nothing inherent in the conjunctive programming concept that restricts linkages only to established portions of program code. Conjunctive programming can serve all phases of the life cycle, because these involve separate, but highly interrelated, products. Life cycle documents include project plans, work implementation plans, statements of work, quality assurance plans, configuration management plans, and various technical documents governing and resulting from the software implementation. Each of these products may contain information that is identical to, or closely related to, information in other products. Productivity and document currency are potentially improved by conjunctive methods applied to the project artifacts.

For example, planning documents often refer to budget goals, obligations, and actual costs; to schedule commitments, milestones, and accomplishments; and to resource constraints, workforce loading, and facility utilization profiles that are planned, monitored, and controlled. These resources are likely to be tracked by a project management system and controlled by the corporate accounting system. Conjunctively programmed project-planning documents can access the actual controlling repositories for accurate and current information, perhaps via interfacing tools.

Requirements documents commonly cite and/or repeat material from governing and auxiliary documents, system documents, user manuals, interface agreements, and results of analyses. Design documents cite material from the governing system and interface requirements, user manuals, outputs of design tools, program design analyzers, etc. All are ripe candidates for the conjunctive approach.

Maintenance documents require detailed as-built design specifications, including construction information in the form of code structure, **MAKE** files, linkage edit instructions, batch scripts, software interface specifications, etc. Section 7 and the appendices give examples of the use of conjunctive programming to record system implementation information.

Planning activities typically create test plans, requirements, criteria, procedures, and test cases, which are set forth and agreed upon before testing actually proceeds. Test results are commonly documented in various reports. Delivery of software to a customer generally requires still more documentation in the form of configuration audits, version descriptions, delivery conditions, installation plans and provisions, and maintenance agreements. Software sustaining and maintenance activities involve implementation of system changes and the maintenance of currency and synchrony among documentation elements and the system components. Potentially, conjunctive methodology can reduce effort and improve the quality of test documentation by reduction of redundancy and maintenance of currency.

2.3 The *ConJoin* Program

The processing engine for conjunctive programs described in this report is *ConJoin*, which, in its present form, is a considerably less grandiose and admittedly simpler tool than **WEB** or **Intermedia**. It is written in C, but could probably have been implemented as an additional macro package incorporated into TEX, had I been a more competent TEXnician. There may even be cut-and-paste programs that are already available on the market that would have saved me the trouble of writing the *ConJoin* program altogether.

ConJoin requires only a minimal production environment: a text editor to create code, data, and documents; TEX, for document production; and a compiler/linker, for translation to machine executable form. It replaces query directives in the conjunctive program file with text copied from other files. Location criteria include the source file name plus either absolute locations within the file, string contexts within the file, or relative locations from string contexts.

I had, at one time, thought of naming the conjunctive programming engine **stitch**, because it belonged in the same genre of names as **WEB**, **TANGLE**, **WEAVE**, and **KNIT**, the mainstays of literate programming. Conjunctive programming, however, is far more than just the "stitching" together of software artifacts, although that is precisely what the *ConJoin* tool does. As in literate programming, the bulk of the effort is not spent in sewing, weaving, and knitting, but in composition. It is with the trusty text editor that the conjunctive craftsperson generates directives that sew extracted software and data elements into place, and it is here that the conjunctive artisan creates the narrative and graphic logic, justification, explanation, and other particulars that give tangibility, meaning, and worth to the finished product. Neither literate programming nor conjunctive programming can be characterized as word processing, even if that is where most of the practitioner's time is spent. In order not to connote the mere mechanics of the tool, but rather the method to which it contributes, I chose the name *ConJoin*.

2.4 Impacts on Method and Expression

Conjunctive programming does impose a discipline on its practitioners. I realized this while developing the *ConJoin* program. My original intent was to provide a method of documentation that did not impact programming at all. I initially wrote a small prototype of *ConJoin* to assist simultaneous development of code and documentation. Other features were appended later, as the needs for them became known. Requirements for additional capabilities were also recognized, some of which are discussed in Section 6.

After adding several features and modifications, I realized that synchronization between the conjunctive source and the accessed files is an essential need of conjunctive programming that is still not yet adequately robust. The programmer is left responsible for creating and maintaining linkages between the accessed code and other data files. No distinctive mechanisms that will be resilient to later changes are automatically provided.

As *ConJoin* evolved, I realized that I was reworking parts of the program that already correctly functioned merely so that I could explain more easily how they operated. Just due to writing about it, the code was changing significantly in structure, although not in function. Additionally, I found many programming simplifications and subtle faults when describing the code. I thus relearned that the way one presents and describes programs influences the way one structures them. It is therefore not totally true that conjunctive programming is completely flexible and conformable to every mode of programming. Some adaptation within the practitioner inevitably takes place.

2.5 Goals

My reason for developing *ConJoin* was pragmatic. I generally document the papers, reports, and programs I write using LaTeX. The style of these documents appears very similar to that of this report. I had just completed a research effort resulting in the development and concurrent documentation of a tool for simulating the software reliability process [20]. In writing the documentation, I manually cut and pasted segments of the code with a text editor into the document. Keeping the document and the program code consistent was very time-consuming and unproductive.

I needed something to extend the environment that I already had, and with which I was familiar—something that would adapt to my method and manner of expression. I did not want to learn another documentation system, nor did I want to reinvent the wheel. I wanted to have the code and documentation separate for ease of editing, compilation, debugging, and distribution. Having a program such as *ConJoin* would have served these needs and eliminated a great deal of frustration and rework.

I began to apply *ConJoin* by using only string-context searching to locate material for pasting into the document being written. I soon recognized the need for more a robust entity reference method as changes were made in the program. Slowly I began to introduce better anchors into the code, but only as needed. Wherever I applied more robust markup principles, the excerpt references proved to be more consistent. Elsewhere, I found that the extractions proved to be more fragile to evolutionary changes in the artifacts. I am still learning the effective means for generating robust, yet convenient, markup.

2.6 Relation to Literate Programming

Literate programs are literal programs; that is, the program source itself is integrally bound within the document that describes it. Every operation and every declaration, however trivial or minuscule, must appear in the document. Literal programs are complete programs, even if narrative, tabular, and graphic descriptions in the remainder of the document are incomplete, inaccurate, and unintelligible. The code appearing in the program document is the actual code that compiles and runs.

Conjunctive programs, on the other hand, do not necessarily document an entire program, line by line. Those portions of code and other data that are displayed within the document are actual and current, as in literate programs. A MAKE program with dependency files maintains currency between the document and the source elements. As in literate programs, descriptions of excerpted system artifacts may be incomplete, inaccurate, and unintelligible.

Both literate and conjunctive programs are (ultimately) processed by TEX, so they both generate documents of high typographic quality. They both permit the creation of textbook-quality documents, when warranted. They both display segments of code interspersed with explanatory text, figures, and tables. The code you see in both documents is the actual program source code. The order of appearance of code segments in both documents is independent of the order in which these segments are presented to the compiler. Both have mechanisms that maintain currency between the program that is seen in the document and that which executes. Both are operated in an edit–compose–render sequence of steps where editing may be interactive, but composition and rendering are batch-mode processes.

Both currently suffer from a number of disadvantages, some of which are technology related:

(1) Each is constrained by the limitations of the document processing system and the program development environment.

(2) Both need better means for displaying figures and graphics, and neither supports a graphics-based design methodology very well.

(3) Neither is oriented for WYSIWYG[3] operation. An interesting experiment would be to adapt ConJoin for use in a WYSIWYG environment, such as Ventura Publisher [21], which also uses a rather simple hidden-text markup that is consistent with the ConJoin program design.

(4) Both have been applied only to traditional types of systems and products. For large programs, document indexes, tables of contents, and lists of figures may not be as useful as online documents with automated display, search, and interaction tools. The current documentation trend is toward distributed heterogeneous multimedia systems incorporating hypertext.

These problems can be overcome by enrichment of concepts, tools, and practices that will take better advantage of the evolving technology.

Conjunctive programming may offer some advantage over traditional literate programming in the following ways:

[3]What You See Is What You Get.

(1) It can adapt to the way people now develop systems. The order of code and documentation development, the separateness of the artifacts produced by these activities, and the content and level of document detail can be tailored to project and individual needs and existing organizational standard practices.

(2) It works conceptually with all programming languages that permit embedded comments and document generators that allow hidden text. Developers already familiar with a documentation system do not have to learn another system.

(3) It is applicable to retro-engineering efforts with no additional risk to the system operation.

(4) It can be used to facilitate concurrent engineering and other forms of collaborative activity, because it can make the documentation and other products of all the teams accessible to everyone without requiring any changes in existing software responsibilities. Programmers can be separate from documentation personnel, and both can maintain cognizance over their separate charges.

(5) It can be used throughout the life cycle, not only for program documentation, but for all kinds of project artifacts that integrate information from multiple sources. Markup standards and change notification procedures can be made a regular adjunct to project implementation plans and practices.

(6) It can use existing development environmental tools, such as symbolic interactive debuggers, text editors, and program analyzers. **ConJoin** does not have to duplicate the functions that commercially available tools already have, but may incorporate their results.

(7) It promotes reuse of existing code, data, and documentation. This reduces "waste" in the Total Quality Management [22] sense.

(8) In cases where the copious display of code in a document is not required or desired, conjunctive programming can just provide specifications, explanation, and a bridge into the pertinent portions of code and data. For example, maintenance personnel claim they prefer reading code in source form once they have an understanding of the program, a road map into the code, and access to explanatory documentation, when needed.

I can also see several disadvantages in conjunctive programming, among which are

(1) There is no automatic assurance of completeness, when such is desired.

(2) There may be a loss of currency in documentation if program elements change without re**MAKE**ing the documentation.

(3) It does not promote structured design and structured programming as well as literate programming.

(4) *ConJo*in does not automatically generate indexes and cross references, nor does it automatically format the resulting document as well as **WEB** does.

3 USING ConJoin

A ConJoin program is a text file formatted for processing by a markup document compiler or generator. As currently implemented, ConJoin programs produce LaTeX documents; other dialects of TeX, such as AMS-TeX and others, can conceivably also be accommodated, although this has not yet been done. This report was produced using LaTeX, augmented by additional style markup commands particular to this report.

Nothing in the conjunctive programming concept mandates the use of TeX or its dialects, but the power TeX provides to the conjunctive programmer offers a considerable incentive. The ConJoin program has been structured to adapt to other TeX-like document-producing engines by run time markup directive redeclarations and recompilable definitions. Adaptation to the UNIX `troff` system or to Xerox's Ventura Publisher, for example, are possibilities for study. TeX is used hereafter to illustrate the character of conjunctive programming as (dis)colored by my own personal preferences, habits, and skills.

Currently, there are only two dependencies in ConJoin on TeX that are not changeable via run time directives. These are the TeX comment-initiation string, "`%`", used by ConJoin to initiate its directives, and `\verb|`*text*`|` used to enclose in line verbatim *text*. Such dependencies are localized in the program as macros that can be redefined to the equivalent commands of another document-generating engine.

3.1 The Source File(s)

The ConJoin tool copies a user's source file, assumed to be an ASCII text file, line by line, directly into a target file. It is not essential that all the characters in the text lines be printable characters, as long as a newline[4] appears regularly within the maximum line length allocation. The lines are assumed primarily to be made up of "regular text," or text that will be recognized and processed by the documentation system (i.e., LaTeX in this report). In addition to this regular text, there are 12 directives that are specially recognized and processed by ConJoin. These all begin with the target-processor comment signal string (for TeX, "`%`"), so that the user source could conceivably be processed directly by the target processor, except that none of the ConJoin directives nor their effects would appear in the compiled document. Rather, the user's source file directs the integration of information segments from many heterogeneous sources into a target file that is then processed by the target processor (TeX or LaTeX, et al.) in the normal fashion.

[4]The meaning of *newline* is generally implementation-defined. Some systems use a combination of carriage return (CR) and line feed (LF) to mark the end of a line, while others use LF/CR or just one or the other by itself, and some have other special conventions. This need not be a problem, as long as the text editor, ConJoin, and the document generator all recognize the same convention.

The length of text lines in the *ConJoin* source file is limited by the value of the `MAX_LINE` macro within *ConJoin* program,

```
        #define MAX_LINE        135
```

Each *ConJoin* directive, except **`%size`**, is copied intact into the target file. Since these are formatted as comments, they will not be seen in the document being created. Each directive, except **`%size`**, should appear on a separate line in the source file, and may not exceed one line in length. The **`%size`** directive may appear anywhere within ordinary text. The *ConJoin* directives are sensitive to capital and lowercase alphabetic characters.

The directives in alphabetic order are:

- `%access` *file_name break_string start_string range_separator end_string*
- `%break` *break_string*
- `%column` *start_column*
- `%count` *count_signal*
- `%garbage` *substitution_char*
- `%path` *directory*
- `%postfix` *end_environment*
- `%prefix` *begin_environment*
- `%range` *range_separator*
- `%show { on | off }`
- `%size { C | a | T | r}`
- `%tabs` *tab_width*

Each of these is described in detail in this section. Path and file names used in the examples are shown in MS-DOS style.

3.2 Selective Inclusion of Text Files: `%access`

The workhorse of the *ConJoin* program is the **`%access`** directive that selectively locates, copies, and formats text from files by using the environment set by the `%prefix`–`%postfix` pair. Most of the material displayed here in the TEX `\tt` font was `%access`-ed from system objects.

For example, the definition of the constant NIL in the *ConJoin* program may be printed by accessing the context of the **#define** statement,

```
%access ConJoin.c, define NIL+0 ~ 1
```

(note that the character # does not precede **define** in the **%access** directive because # is the default *count_signal*, described later). *ConJoin* responds with the result

```
        #define NIL             ((void *) 0)
```

ConJoin supports three kinds of location markup tagging:

(1) by context (as above)

(2) by line position in the file

(3) by line position relative to a context

This markup recognition is not as general as that offered by a HyTime-compliant engine, and may be augmented in future versions (see Section 6). *ConJoin* can find contextual keys in the system database, either occurring naturally or placed there as anchors for more robust access. As a result of my using both approaches in describing the *ConJoin* program in Section 7, I now recommend that users embed unique, standardized anchors in the system database that will withstand system evolution. However, all three reference schemes will be explained below.

The syntax of the directive is

%access *file_name break_string start_string range_separator end_string*

The *file_name* identifies the name of the file in which the text segment to be copied may be found. The *break_string* is "," by default. The optional *start_string* and *end_string* take the form

[*match_string*] [*count_signal count*] [±*offset*]

In each case, the *match_string* is a set of substrings, separated by *break_string*, of characters to be matched exactly (case sensitive) within the named file, in order. That is, matches for each substring are sought in the order given. Subsequently, the *offset* is an integer number of lines past or before the string match condition. The *count*, when present, is always considered positive and denotes the number of matches necessary to start action. The default *count* is one. The number of substrings is limited in number to

```
        #define MAX_CONTEXT     10
```

If the starting *match_string* is missing, then the match condition is satisfied at the beginning of the file. A default offset of unity is presumed if the starting *offset* is missing. A unity offset causes all lines up to and including the matching line of the named file to be skipped. Setting the offset to +0 prints the line fulfilling the string match condition, and a positive starting offset of *n* causes *n* lines in addition to the matched line, to be skipped.

Negative offsets are also permitted. Setting the beginning or ending offset to -*n* causes the action to take place *n* lines before the fulfillment of the string match condition. It is necessary, in this case, that *ConJoin* maintain a queue of length *n*, which is limited in size to

```
#define MAX_Q            100
```

If the ending *match_string* is present, but the *offset* is missing, an offset value of -1 is assumed. *ConJoin* stops copying at the line matching the *match_string* condition in this case. Setting the ending offset to +0 causes the line fulfilling the *match_string* condition to print.

If the ending *match_string* is missing and the ending *offset* is non-positive (the default when missing is -1), copying extends until *offset* lines prior to the end-of-file. If the ending *match_string* is blank, but a positive *offset* is present, that value specifies the number of lines that will be copied from the named file.

The + in a positive *offset* may be omitted when the *match_string* and count fields are absent. The - in negative offsets must always appear.

The range separator default ˉ between the beginning and ending extraction keys above can be replaced, if desired, with a more convenient mark by using the `%range` directive described later in Section 3.9. The *count_signal* default `#` that signals the beginning of the *count* field may also be changed by using the `%count` directive described in Section 3.5.

A typical usage of the `%access` directive is illustrated in the following:

```
%access ConJoin.c, * strext *+0 ˉ * end strext *+0
```

which produces

```
                                                                  /* strext */
/***************************************************************************/
        STRING
strext(s, t)    /* Extract string t up to the close-comment string, if close
                   is not null, or to the end of t if null, into s. Remove
                   leading and trailing blanks from s and return s.
/*------------------------------------------------------------------------*/
STRING  s, t;
{
        STRING p;

        if (*close AND (p = strstr(t, close)))
                strtcpy(s, t, p - t);
        else
                strcpy(s, t);
        return stratrim(s);
}
                                    /* end strext */
```

As seen above, this usage copies all lines between the first occurrence of a line containing the beginning markup anchor substring **`* strext *`** and the next occurrence thereafter of a line containing the ending markup anchor substring **`* end strext *`**, inclusively, into the target file. The result in the target file is the source code for the **`strext()`** function shown below and explained more fully later in Section 7.19.

Insertion of **`%access`**-linkages into the code in the form of distinctive comments, such as **`* strext *`** and **`* end strext *`** above, brings a measure of robustness that is not found with other contextual means of location. Synchronization among conjoined files and **`%access`** directives is made more reliable when separate markup is provided. Other means of text segment location, such as by match count and offset, are much more fragile to changes in and movement of code functions. Section 6 discusses possible future enhancements in conjunctive programming tools that will promote further synchrony.

When the `+0` *offset* designations in both the beginning and ending search strings are omitted, the result is

```
/************************************************************************/
        STRING
strext(s, t)    /* Extract string t up to the close-comment string, if close
                   is not null, or to the end of t if null, into s. Remove
                   leading and trailing blanks from s and return s.
/*----------------------------------------------------------------------*/
STRING  s, t;
{
        STRING p;

        if (*close AND (p = strstr(t, close)))
                strtcpy(s, t, p - t);
        else
                strcpy(s, t);
        return stratrim(s);
}
```

The file segment could have been displayed just as well by using a negative beginning offset,

```
%access ConJoin.c, strext( #9 -2 ~ 16
```

or using an ending search string that stops 3 lines earlier than the next function-header-line,

```
%access ConJoin.c, strext( #9 -2 ~ ***** #2 -3
```

because the **`strext()`** function appears on the ninth appearance of the name in the program file. This method is not recommended, as it is very sensitive to future changes in the program code.

Multiple substring context matches can also be specified as a means to avoid ambiguity in locating a desired point in a file. As an example, the appearance of **`close`** in the code segment above can be located by first searching for

`* strext *`, then bypassing the appearances of **`close`** in the function description by looking for **`STRING p`**, and finally, by seeking the next appearance of **`close`**. The **`%access`** directive is written

```
%access ConJoin.c, * strext *, STRING p, close +0 ~ 1
```

which results in the single line

```
                if (*close AND (p = strstr(t, close)))
```

Note: If a *match_string* contains either + or -, then it is necessary to apply the *count_signal* so as not to mistake the sign character for the start of an *offset* count. For example, to access only the function banner portion of the **`strext()`** module, one could use

```
%access ConJoin.c, * strext * ~ /*--- #1 +0
```

3.3 Setting the Context Separator: %break

The default delimiter for file names and *match_string* in the **`%access`** directive is the comma character, "**,**". This can be changed by using the directive

`%break` *break_string*

The default is thus equivalent to the statement

```
%break ,
```

The *break_string* may contain any (non-null) characters, including embedded white space, but any leading and trailing white space is removed. The *break_string* and the *range_separator* must be different (for error detectability reasons), and these must differ from the *count_signal*, +, -, and all characters that appear in the *match_strings*.

3.4 Setting the Alignment Column: %column

If, at times, the **`%access`**-selected text has too little or too much white space to the left of the lines to be printed, an alternate starting column can be selected by the directive

`%column` *start_column*

The *start_column* may be positive (added spaces) or negative (deleted characters). The default starting column is zero, so the default is equivalent to the directive `%column 0`.

As an example, the ConJoin program contains a code segment that appears in the `ConJoin.c` file as

```
                    if (NOT (txqueue[qex] = strdup(text)))
                    {       error_message(MEMORY_ERR, "", FALSE);
                            error = TRUE;
```

```
                        break;
                    }
```

Using a %column -16 directive causes the output to take the form

```
if (NOT (txqueue[qex] = strdup(text)))
{       error_message(MEMORY_ERR, "", FALSE);
        error = TRUE;
        break;
}
```

The alignment column remains in effect until reset by another **%column** directive.

3.5 Specifying the String Match Count Signal: %count

The beginning and ending **%access** string matching specifications may, at times, themselves contain the default match-count-signal, **#**. In such cases, the match count signal may be changed by using the **%count** directive

%count *count_signal*

The default is thus equivalent to

%count #

The *break_string* and the *range_separator* must be different (for error detectability reasons), and these must differ from the *count_signal*, +, -, and all characters that appear in the *match_strings*.

Examples of the use of the *count_signal* appeared earlier, in Section 3.2.

3.6 Conversion of Non-Compilable Characters: %garbage

Text files occasionally contain non-ASCII characters (such as those of the IBM character graphics set) that cannot be printed by the target processor. In such cases, *ConJoin* makes a substitution. All non-printable characters are replaced by a "garbage" character, which by default is **#**. This default can be changed by using the directive

%garbage *substitution_character*

The *substitution_character* remains in effect until reassigned by a later **%garbage** directive. If no substitution character is named, no substitution is made. No provision is currently available in *ConJoin* for character translation other than this simple indication of where non-printable characters have been encountered. The default is equivalent to

%garbage #

As an example, all the files describing and comprising *ConJoin* contain a copyright header that contains graphic characters recognized by most IBM-compatible printers, but not by TEX. When *ConJoin*-ed, this banner appears

```
%*************************************************************************
%*                                                                       *
%*        Copyright (C) 1992, California Institute of Technology         *
%*      All rights reserved.  U. S. Government sponsorship under NASA    *
%*                  Contract NAS7-918 is acknowledged.                   *
%*                                                                       *
%*                       Robert C. Tausworthe                            *
%*                     Jet Propulsion Laboratory                         *
%*                       4800 Oak Grove Drive                            *
%*                     Pasadena, CA 91109-8099                           *
%*                                                                       *
%*                                                                       *
%*************************************************************************
```

When viewed by a text editor or as a direct-dump printout, the garbage characters above form a double-line box around the notice.

3.7 Setting File Search Paths: %path

When text segments to be accessed are in a directory other than the current default path, or if segments are to be copied from files in several directories, these paths may be identified by using the %path directive,

%path *directory*

Paths remain in effect throughout the remainder of the *ConJoin* file.

Any number of path directives may appear, up to the maximum number defined in the *ConJoin* program, currently

```
#define NUMBER_OF_PATHS  20
```

The benefits of the %path directive are that its usage can help make document files more portable and adaptable to multiple platforms, permitting localization and concentration of search directory specifications, thereby reducing the sensitivity of the document to path-naming conventions. Furthermore, it also shortens file names appearing in %access directives.

An example of the %path directive usage is the following: Several functions accessed by the *ConJoin* program are located in a separately compiled library for which the source code happens to be available. This code appears in the directory \c\topc\c, which, as one may easily guess, refers to "Tausworthe's Own Personal C" library (TOP-C).

```
%access \c\topc\c\stratrim.c, *****+0 ~
```

Alternately, one may use a %path statement to identify the library directory,

```
%path \c\topc\c\
%access stratrim.c, *****+0 ~
```

Note that the final directory separator (here, \) is necessary in the %path statement, as path strings are directly concatenated with the accessed file name in searching for the file.

Either of these two alternative forms transcribes the same file segment into the target document:

```
/****************************************************************************/
        STRING
strtrim(s)      /* Trim all white space from s, leading and trailing, and
                   then return s.
/*------------------------------------------------------------------------*/
STRING s;
{
        FAST STRING p;

        p = strfnb(s);
        strtcpy(s, p, strlen(p));
        return strtrim(s);
}
```

3.8 Setting the Environment: `%prefix` and `%postfix`

The environment for printing the text selected by the **`%access`** directive is controlled by **`%prefix`** and **`%postfix`** directives:

> **`%prefix`** *begin_environment*
> **`%postfix`** *end_environment*

where the beginning and ending *environment* portions of the directives are target-processor commands that set up and terminate the environment for printing **`%access`**-ed text. The defaults are equivalent to the following directives, which enclose the LaTeX **`verbatim`** environment:

> **`%prefix {\footnotesize \begin{verbatim}`**
> **`%postfix \end{verbatim}}`**

These defaults print the intervening text by using the **`\tt`** font, sized small enough that an 80-character line fits into the width of a printed page. Lines appear in the output document exactly as they do in the text file. *ConJ*oin does not perform "pretty printing" of the selected text. The regular font style and size environment are restored by the **`%postfix`** statement.

Selected beginning and ending environment statements stay in effect until changed.

3.9 Defining the Access Selection Separator: `%range`

The default ~ separator appearing in the description of the **`%access`** directive above may not always be effective to use, such as when a desired starting *match_string* contains a ~ character. The default may be overridden by the directive

> **`%range`** *range_separator*

All leading and trailing white space in the *range_separator* are discarded by

ConJoin. The *range_separator* remains in effect until changed by another **%range** directive. The default is equivalent to

```
%range ~
```

The *break_string* and the *range_separator* must be different (for error detectability reasons), and these must differ from the *count_signal*, +, −, and all characters that appear in the *match_strings*.

3.10 Displaying the ConJoin Directives: %show

ConJoin directives, except **%size**, can be displayed in the target document by using the **%show** directive,

```
%show { on | off }
```

When on, each occurrence of a ConJoin directive is made visible in the target output file. The action of the directive takes effect immediately.

The directives used to display the **NUMBER_OF_PATHS** definition in Section 3.7, for example, were

```
%show on
%column 8
%prefix {\begin{verbatim}
%access ConJoin.c, NUMBER_OF_PATHS+0 ~ 1

        #define NUMBER_OF_PATHS  20

%column 0
%prefix {\footnotesize \begin{verbatim}
%show off
```

The display setting persists until reversed by another **%show** directive. The default is equivalent to

```
%show off
```

3.11 Displaying File Sizes: %size

Each time ConJoin is run, a file having the same name as the target file, but with a **.siz** file type, is written that tells the total numbers of lines

(1) in the Conjunctive program source file

(2) transcribed into the target file by **%access** directives

(3) written to the Target file

(4) the ratio of the latter two

These numbers are available via **%size** directive, which may appear anywhere within a line being processed. The syntax is

```
%size {C | a | T | r }
```

The options, C, a, T, and r may appear either capitalized or uncapitalized and, respectively, correspond to the four items above. Since the items refer to the sizes of the file when it was last processed, there may be an error if the number of lines has changed. To avert this possibility, *ConJ*oin should be run at least twice before processing by TEX.

As an example, **%size** is used four times in the following sentence where underlined: The last time *ConJ*oin processed the **CJ_body.CJn** file, it contained 3159 lines, which transcribed 1184 lines from other files into the target file, to produce a total of 4351 lines in the target file. The extra 8 lines in the target file over its constituents is an identification banner, described later in Section 7.15. The ratio of transcribed-to-total lines is 0.272.

3.12 Altering the TAB Width: %tabs

By default, tab characters in the **%access**-selected file are expanded to align text on columns 8 characters wide. This default may be overridden by using the **%tabs** directive,

%tabs *tab_width*

The default is thus equivalent to

```
%tabs 8
```

The TOP-C library function **strtcpy()** with **%tabs 4** in effect displays as

```
/************************************************************************/
    STRING
strtcpy(s, t, n)     /* Truncated string copy.  Copy at most n CHARs
                of t into s, and return s.  Note:  in contrast
                to strncpy(), the returned copied s always ends
                in NUL.
/*----------------------------------------------------------------------*/
STRING      s;
const STRING    t;
size_t      n;
{
    size_t m;

    if (s)
    {   m = strlen(t);
        n = MIN(n, m);
        memmove(s, t, n);
        *(s + n) = NUL;
    }
    return s;
}
```

The same code with the default **%tabs 8** in effect appears later in Section 8.7 for comparison.

The tab width remains in effect until altered by another **%tabs** directive.

4 INSTALLING AND RUNNING *ConJoin*

This section describes how to run *ConJoin* on the IBM Personal Computer, or compatible. It does not explain how to run either TEX or LATEX or the C compiler. It is assumed that user guides are available for these and that the user is already familiar with their operation. To date, *ConJoin* has only been implemented for the IBM PC; however, the program is written in ANSI standard C, except for a few routines that have been accessed from my TOP-C library, which are included in a separate source file on the product disk.

4.1 Configuration

The *ConJoin* system described in this report requires the following environment:

(1) An IBM Personal Computer, or compatible device with at least 128 kilobytes of available RAM for execution and utilizing the MS-DOS operating system Version 3.1 or later.

(2) About 100 kilobytes of available disk or diskette space for system storage. Disk storage is preferred, and the system has not been tested for floppy diskette operation. Installation in the next section is limited to hard-disk configurations.

(3) TEX and LATEX (or other publishing system), text editor, and other documentation tools, with the peripherals and storage they require.

(4) If modifications of the system are to be made, a text editor, C compiler, and linker. The `.bat` and `.mak` files provided on the product disk are configured for Microsoft C 5.1, and for my directory structure and development system. These may have to be edited to conform to other compilers and user environments. Additionally, the `top.c` functions should be appended to the `ConJoin.c` file prior to compilation. See Section 9 for more information on program modifications.

4.2 Installation

The steps for installing the *ConJoin* system are:

(1) Create a (preferably separate) subdirectory, such as `\conjoin`, and make this the default directory:

```
>md \conjoin
>cd \conjoin
```

(2) Copy all the files from the distribution medium into this directory. For example, if the distribution is a disk in drive `a:`, then

```
>copy a:*.*
```

(3) Invoke ConJoin with no parameters to see the usage message:

```
##########################################################################
#                                                                        #
#        Copyright (C) 1992, California Institute of Technology         #
#      All rights reserved.  U. S. Government sponsorship under NASA     #
#                  Contract NAS7-918 is acknowledged.                    #
#                                                                        #
#                         Robert C. Tausworthe                           #
#                       Jet Propulsion Laboratory                        #
#                         4800 Oak Grove Drive                           #
#                       Pasadena, CA 91109-8099                          #
#                                                                        #
#                                                                        #
##########################################################################

                             ConJoin Program
                              (08-Apr-1992)

Usage:  ConJoin <ConJoin source> <target file> [<options>]

        Source file type default is .CJn
        Target file type default is .TeX

Options:
        -a       Do not announce the program.
Command line error: No source file named.
```

This, or a similar message also appears whenever ConJoin detects a condition under which it cannot proceed further.

4.3 Running ConJoin

There may be differences in running ConJoin, even on the IBM PC, or compatible, primarily in how execution is initiated and how files are specified. The description here outlines the operation in *batch mode*. Once ConJoin is initiated, it reads its input files and writes its output file until completion. Selection of the files is made on the command line at the DOS prompt. Some implementations may have menu- or window-based user interfaces that alter this procedure somewhat. The user is expected to know how to create the equivalent of a DOS command invoking ConJoin and specifying its input, output, and options within the user interface of the platform involved.

(1) **File naming**. The DOS operating system locates files by subdirectory "path" and by "file name" and "file type" (or "extension") within a directory. The user is expected to understand these general conventions, as they are not further explained here. Files input to and output by ConJoin may be specified to have any subdirectory, name, and type. A missing

type in the input file specification defaults to `.CJn`; a missing output file type specification defaults to **`.TeX`**. *Conjoin* creates a file with the same directory and name, but with type **`.siz`** that contains processing size information (see Section 3.11 for further information on access to this information). *Conjoin* also saves the previous output file, if one existed, by giving it a `.T_X` extension and removing the old `.T_X` file.

(2) **Building source (text) files**. The generation of a TEX conjunctive program is almost the same as writing any other TEX document, except that when information from a system data file is needed, or if the information to be accessed is to appear in a modified format in the resulting document, then at these points, *Conjoin* directives are used as heretofore described.

To ensure that material be more easily or more robustly transcribed, the user may edit the source file to insert entity reference markup in the form of comments. Entity reference markup considerations are discussed further in Section 5.

(3) **Execution**. While *Conjoin* may be operated from any directory as long as fully qualified path names are used to locate *Conjoin*, source, and target files, it is generally more convenient to change the default directory to that in which the files are found,

```
>cd path
```

Then invoke *Conjoin* to process each conjunctive program by using

```
>c:\conjoin\conjoin options source target
```

The *options* may appear anywhere within the invocation, but the *source* must precede the *target* designation. The *source* must always be present, but *options* and *target* are optional.

If *target* is omitted, a default name is generated by using the *source* name, but changing the file type to **`.TeX`** (current default).

Only one *option* currently exists: **-a**, which causes the JPL/Caltech copyright announcement, program name, and version date to be omitted. Possible future options are described in Section 6.

5 FURTHER CONSIDERATIONS

The continual, elaborate, and extensive flow of electronic documents within and throughout industry, universities, and government agencies has created version, review, revision, and currency problems for nearly everyone involved. Authors often create multiple versions of a document for different purposes (full report, literature paper, executive summary, etc.), to be communicated broadly and to be viewed via e-mail, printout, video, and typeset media. When comments and responses received from multiple sources must be matched to the versions reviewed, all the information communicated must be managed in an orderly way. This section discusses three areas where care and thought must be applied in the conjunctive documentation life cycle: organization, linkage markup, and retro-engineering. I deem these areas important because of the lessons I learned in developing this report.

5.1 Document Organization

This publication, in its current form, is the integration of many files conjoined for TEX processing. The top-level master document file shown in Appendix B.1 controls the integration of the many separate, constituent components that make up the report. Appendix B.3 shows how the call-tree of Appendix C.1 and the reference list of Appendix C.2 are made each time the `ConJoin.c` program changes. The document `MAKE` file in Appendix B.6 directs the generation of the program size file, the translation of `.CJn` forms into TEX files, and the subsequent production of the report by TEX.

The report structure is formed as it is because other documents are, or are planned to be, constructed from the same baseline files by inclusion, exclusion, and `%access`. The markup and composition techniques that have been employed did not originate with this aim in mind, however, and for lack of my attention, do not yet entirely fulfill this goal. Some rules for promoting currency and synchrony have been relearned and reappreciated during the writing of this report:

(1) Design the documentation using object-oriented concepts. Organize material into separate, cohesive objects. Establish classes of items within the system that will promote stability of products during an evolutionary life cycle. Access entire objects to the extent feasible.

(2) Plan to develop separate volumes of product documents from a common database of information from the beginning. Design the documentation, code, and data schemata to accommodate multiple views (subschemata) of product documents.

(3) Separate document style design from document content design.

(4) Use aliases in source documents that will substitute application-specific expressions in output documents. Localize the definitions of aliases into a separate object for that application. In particular, use aliases for all "magic numbers," or arbitrary constants that might eventually change over time or across documents. As an example, a LaTeX command `\work` was created to denote the current document type, here "report." Every appearance of the word report in this document traces to an appearance of `\work` in the source file.

(5) Use syntactic and semantic location mechanisms to refer to internal document structural entities (sections, subsections, bibliographic citations, etc.), rather than absolute positional information.

(6) Use standard document types, outlines, and templates, when available. This often prevents having to "reinvent the wheel" when designing the documents in a new project.

5.2 Entity Reference Linkages

The successful evolution of conjunctive programs hinges on the query techniques used to access information from system code and data media. All conjunctive programming engines must support persistent tagging of information for query. (In hypertext and HyTime, linkage tags are called *anchors*.) Code units to be located by queries range in hierarchy from directories, to files, to functions or declarations within files, to clauses within them, to statements or lines within clauses, and finally, to atomic units (numbers, words, or other tokens). Similar hierarchies apply to other forms of information. Synchronism between copied information and the document context tends to decrease as the granularity increases. That is, the content of 4 lines of text starting at the first occurrence of the token "5" in a particular file is apt to be much more contextually variable than is, say, an entire function. It is true that the function may change over time, but the document context probably still concerns that function, whereas the mere change of an earlier parameter value from 4 to 5 in the former case completely destroys the correspondence between the code and the document context.

Queries may be made by using absolute or relative locations, contextual information, semantic content, or a combination of these.

Linkages to items in an information base may be made on the basis of absolute and relative locations, context, and semantic content, in order of increasing robustness. Each form of link requires some form of query processing. For absolute and relative locations, mere counting suffices. In the case of context queries, string matching or pattern recognition may be applied. Searching for semantic content requires a system capable of interpreting the data it encounters. *CorJoin* accommodates the first two types of linkages in the form of an absolute count of sequences of matched patterns followed by relative count of lines.

Linkages to textual items can be made purely on the basis of location and context information alone if desired. However, the use of distinctive markings within the data purely for the purpose of establishing such linkages can produce a much more robust synchronization between the referenced information and the document context. Inserting linkage tags in code and data files may require agreements with the owners and/or managers of those objects. However, the minimal extra effort required during implementation to insert these marks may yield a significant payoff in documentation productivity later.

Plans for making robust linkages should be considered early in the product life cycle. Projects or cognizant individuals may need to develop standard conventions for linkage markup, or may apply standards, if available. The tagging should be distinctive and recognizable as linkage markup. It should have both beginning and ending delimiters (these are useful for location, modification, and removal, when needed).

Cohesive segments should be tagged whenever it is likely that access will be made and location by the surrounding context is unreliable. In data where embedded markup is disallowed, such as in data managed by a database management system (DBMS), other provisions may have to be made. For example, preprocessing the information using queries to a DBMS may be necessary. Alternatively, entity reference tags may be put into an auxiliary file and separately accessed via a special linkage engine.

Robust markup works in both directions. Besides helping to synchronize working files and documentation, recognizable linkages help during maintenance in locating all the places in the documentation where descriptions of information may be accessed. When code changes, for example, the linkage provides clues for finding the corresponding narrative. Without recognizable markup, one is left with a somewhat more difficult search. When changes are made in a program file, one must search through all `.CJn` files for **`%access`**-es bearing the altered file name. Then, in those files, one must look for contexts consistent with material that was in the file before the changes took place. Some of this may be made easier by a utility program to print file differences, as a guide to where the changed areas are, and a utility program to trace **`%access`** conditions into the code to check whether the range intersects one of the changed areas.

With distinctive linkage tags, however, one need only search for **`%access`**-es to the changed files bearing the matching tags in the areas of change.

I had been working on the concept of conjunctive programming and the design of *ConJ*oin for some time when the HyTime article appeared in the *Communications of the ACM* [8]. I recalled while reading it that I had earlier reviewed a draft of the proposed Standard Generalized Markup Language for JPL a few years before and had since forgotten all about it. The ACM article made me suddenly realize that conjunctive programming is part of a much wider data interoperability discipline, one for which standards are emerging into practice.

Future work in conjunctive programming should investigate the use of the international standard for SGML [9] in conjunctive programs and augment *ConJoin* accordingly.

5.3 Retro-Engineering

The terms "retro-engineering," "reverse-engineering," and "re-engineering" have been assigned slightly different meanings by researchers and others, but those terms generally refer to efforts to redevelop quality attributes within existing products after they have been implemented. I will not distinguish among the subtle differences here. The process is one that involves recovering or improving the design and translating, restructuring, or augmenting the program code. Tools exist in some environments to create portions of the new documentation—in narrative and diagrams—directly from scanning the code, while other tools assist in converting the code into required forms. Documentation educed from source code does not generate any new information about the code, although it may present data about the code in a more human-understandable form. Code translation can be slow, error-prone, inefficient, and costly.

Problems in retro-engineering tend to fall into one of the following areas:

(1) Implementation bias—recovery of general design information, rather than language and system considerations, from available information, is difficult and requires careful analysis.

(2) Traceability—links between recovered information and original sources is needed for tracking completeness, consistency, and fulfillment of objectives.

(3) Domain knowledge—the purpose, precision, range of values, rationale, theoretical basis, and significance of entities are missing and must be recreated.

(4) Code correctness—latent faults may be duplicated into the re-engineered products.

(5) Information credibility—defective comments and documents may be used to re-engineer products, and faults in the code may make documentation untrustworthy.

The tools available for retro-engineering are generally the same as those that support forward engineering. They can produce data flow diagrams, control flow diagrams, structure charts, data structure diagrams, entity-relation diagrams, state-transition diagrams, and online dictionaries, and they can produce documents, analyses, and measurements. However, even though efforts may be significantly assisted by the use of automated tools, retro-engineering remains largely a human task of supplying information, structure, and capability not otherwise derivable from existing products.

Conjunctive programming can be useful in generating traceability links, recording recovered design and domain knowledge, conjoining appropriate portions of original and newly developed artifacts, and preparing the documents required. It can contribute to productivity by recording the recovered design by using media specifically developed for handling documents and linkages among information entities. Conjunctive programming should be particularly effective in efforts involving redocumentation only, because existing artifacts may not have to be altered at all.

6 FUTURE USAGE ENHANCEMENTS

The simple capabilities of ConJoin discussed in Section 3 merely hint at the utility of tools in support of conjunctive programming. The present section discusses additional features and tools that may be developed if the ConJoin prototype is successful in garnering the attention of users at JPL or in NASA, other government agencies, or industry. The order in which these, or other functions suggested by users are developed, will be dictated by the needs of users, development costs, and availability of resources. Estimated effort for making the changes below also includes resources for review, test, and documentation. Costs estimated as "minimal" are expected to require a maximum of one workweek of effort.

(1) A directive that is the same as **%access**, but which does not insert prefix and postfix strings: **%use**. This capability is available currently in ConJoin, but is awkward, because the prefix and postfix must be nulled before and reset after using **%access**. Cost: minimal, a few lines of code and 2 subsections of user-guide documentation.

(2) Provisions to permit matching leading and/or trailing blanks of entity reference strings. Cost: minimal, a few lines of code and alterations in the user's guide.

(3) Directives **%open** and **%close** to change comment delimiter strings. Cost: minimal, a few lines of code plus a new section added to the user's guide.

(4) Directives **%preshow** and **%postshow** to alter in-line verbatim command prefix and postfix strings. Cost: minimal, a few lines of added code and a new section added to the user's guide.

(5) A **none** option for the **%show** directive to prevent the copying of ConJoin directives into the target file. Cost: minimal, a few lines of code and a short addition to the user's guide.

(6) A directive **%ignore on | off**, or other such means to disable a directive without having to delete it. Cost: probably minimal, with only a few changes to ConJoin and user documentation, but some thought must be given to scope, exact syntax, and selection of ignored items.

(7) Directives and command-line options to change all defaults, with an option to read command-line arguments from a file. Cost: minimal, a few lines of code each and a few corresponding changes to the user's guide.

(8) Directives for selective access to contents: directives, such as **%includeif** *condition* or **%excludeif** *condition*. Details on just how this should function must be worked out, especially with respect to the form of the logical *condition*. Cost: unknown, but probably could be completed in less than

three workweeks, one to work out details, one to make the changes, and one to develop the user's guide material, if they prove to be useful and feasible.

(9) Extension of the `%size` directive to permit access to other `.siz` files. Cost: unknown, but probably would require two workweeks, to work out details, develop code changes, and amend user's guide material.

(10) A means to reinstate initial defaults. Cost: minimal, a few lines of code, plus minor additions to the user's guide.

(11) Capability to push prefix/postfix pairs on a stack and to pop the stack back to a previous environment, or to define multiple environments. Cost: unknown, but probably would require less than 2 workweeks to work out details, develop code changes, and amend user documentation.

(12) Better execution efficiency through slight algorithmic changes, such as retaining open files to be accessed again, better pattern searching than from the beginning of the file each time, and creating variables to replace the various reevaluations of `strlen(xx_SIGNAL)` that appear throughout the program. Cost: unknown, but incremental improvements would probably require less than 2 workweeks each.

(13) Improvement of program tolerance to changes by improving the cohesiveness of functions and the removal of side effects. Some candidates are discussed in the Internal Operations Section, below. Cost: unknown, requires a more thorough evaluation of likely maintenance traffic.

(14) Means for conjoining nested accessed files to an arbitrary depth. Currently, only one level of conjunctive commands is accommodated. This feature would permit text segments accessed by *ConJoin* to contain further *ConJoin* directives. Cost: unknown, but would probably require less than 2 workweeks.

(15) Directives to access the date and time stamps of files, and to make decisions based on these. Cost: unknown, requires more refinement of functional requirements.

(16) Command line options to set all *ConJoin* defaults. A command line option to input all options from a file; an option and directive to prevent *ConJoin* directives from being copied into the `.TeX` file. Cost: minimal, only a few lines of code and minor changes and additions to the user's guide.

(17) Extensions to accommodate other document-producing systems, such as Ventura Publisher, and others. Cost: unknown, some study will be required, but probably would require less than 3 workweeks, if feasible.

(18) A more integrated system of tools that would permit automated generation and maintenance of anchors and linkages, as in [10]. Cost: comparatively high, as initial acquisition costs of such a system are unknown.

(19) A linkage manager that automatically registers the usage of conjoined segments of files and oversees the maintenance of anchors and objects within files.

The progression from a simple tool, such as ConJoin, to the ideal conjunctive programming environment would require a more formal systems engineering effort and a significant commitment of programming resources. Cost-effectiveness would probably be reached far short of the ideal system, after having acquired a set of tools that bridge the major difficulties in document creation and maintenance. Some useful auxiliary tools that come to mind include

(1) A coverage analyzer to assess the degree of completeness with which the conjoined document describes an entire program or set of programs.

(2) A markup tagger with features for automatically generating robust entity reference anchors within programs and other files.

(3) A tool to analyze programs and other files and to generate candidate linkages that should be made within a conjunctive program.

Additionally, future efforts may convert the ConJoin system database into a form conforming to SGML and HyTime standards or to integrate with hypertext systems, such as Intermedia.

7 INTERNAL OPERATIONS

This section describes the *ConJoin* program in a form as literate as is possible for me to produce in normal documentation time. The goal in normal documentation time is not necessarily to be literate, but descriptive and communicative to an audience with assumed skills and levels of expertise. In describing the program below, I have assumed that the reader is familiar with the ANSI standard C language and its library functions. I will not explain the C statements nor the ANSI library functions, aside from their roles in the *ConJoin* program, when noteworthy.

The *ConJoin* program itself does not currently fully conform to the SGML markup standards but does align with their goals.

It is difficult without referees to know how much explanation is necessary for describing a program, even to an assumed audience, particularly one as short as this, having only 977 lines[5] of code. Is it necessary, for example, to tell a programmer why a C program includes `stddef.h`? Every C programmer is familiar with this header file and more likely has to be told if and why it has been left out, rather than why it has been included. Understanding what a program does and how it does it generally requires less information than the literate program, which must document every detail of compilable matter.

However, for maintenance and reuse, it is necessary to know what use is being made of all information given to the compiler. For example, the *ConJoin* program at one time during its development referred to `INT_MAX`, a value defined in `limits.h`. A subsequent design improvement deleted that reference, whereupon it was possible to remove the corresponding `#include` statement from the code. Neither reader understanding of the functions and algorithms nor the computer performance was impaired by the unneeded `#include` in the program. Nevertheless, it was superfluous, outdated, and proper for exorcism by the attentive maintenance programmer (viz., me).

In the remainder of this report I shall attempt to describe what I think a reader fluent in ANSI C should know in order to understand precisely what the *ConJoin* program does, how it does it, and why it was constructed as it was.

My particular manner of expression will be evident throughout the program. Although certainly influenced by the Plum Hall standards [23] of the early 1980s, my own style has evolved into a fairly consistent, somewhat distinctive set of practices and habits summarized below. Some may find fault with the form and composition, others will not. Practically all will notice, however, that it is distinctive. Whether this style would be effective if used by others is unknown. A bibliography of research in programming style appears in [24].

The point is that conjunctive programming and *ConJoin* have permitted

[5]This number was `%access`-ed from a file `CJ_prog.siz` using empty prefix and postfix strings (i.e., in a non-verbatim mode). The file was written by a utility program `flines` that scans the *ConJoin* program and records its length whenever *ConJoin* changes. This occurs automatically, directed by `make`.

PRECEDING PAGE BLANK NOT FILMED

42 INTENTIONALLY BLANK

me to express myself in text and in code in the way I wanted[6]—for better or worse. The way I indent code in the source files is the way it appears here. Tortuously convoluted operations that appear nested within C `if`-constructions and elsewhere reflect the way that I think in code. I need documentation to help me after a while to unravel the intricacies of expression and to recall what I must have been thinking at that time—to revive the latent intellectual character of what would otherwise appear to be cryptic.

As algorithms go, none in *ConJoin* is particularly curious. In fact, every function seems perfectly straightforward—except for a few subtleties here and there that I hope will be clarified. My optimism is probably natural, since I have just recently written the program and it is still fresh in my mind. To become more sensitive to what this report should have contained, but does not, I will need feedback from others and an opportunity to redo it in a year or so, to provide the rationale now seemingly too obvious to be mentioned (but which will likely be evident by then), to correct misstatements, to include informative material obtained after publication, and of course, to update the narrative with descriptions of new and altered features.

The degree to which the reader comprehends this report will measure the extent to which I have been successful in attaining my goals using conjunctive programming. Insofar as those goals have eluded me, I hope the reader will find the concepts and approach informative, or at least curious.

The code for *ConJoin* is contained in the file `ConJoin.c`, and is listed fully in Appendix A. The program displayed in this report should be viewed as the internal JPL prototype to illustrate conjunctive programming. Some of the enhancements discussed in Section 6 will probably appear in versions eventually released for wider usage.

7.1 A Word on Programming Style

This section is included not to defend my programming style but to explain what will be seen. The normal order of items in one of my program files is

(1) Program header, with version, file name, copyright, and author declarations.

(2) Header files, in order as applicable: ANSI standard headers, ANSI-conformal library and macro headers, special system library and macro headers, and, lastly, system-dependent library and macro headers.

(3) Local macro definitions, if any.

(4) Global function prototypes not contained in include-files, if any.

(5) Global data structures, if any.

[6]This is not so surprising, since I developed *ConJoin* to do what I wanted. Hopefully, however, I have made it general enough to permit others this same freedom.

(6) Local function prototypes, if any.

(7) Local data structures, if any.

(8) Functions, in alphabetic or depth-first[7] order, except for **main()**, which is always the first function.

Each of these is displayed in a distinctive way. For example, the declaration of each function appears inside a banner (see the **strext()** function in Section 7.19) that consists of a right-justified comment containing the function name, followed by a distinctive, eye-catching row of asterisks. The function scope and return-type declaration appears on the next line, indented to emphasize the function name and parameters on the next line, flush left. An annotated description of the function and its return values, followed by a row of hyphens to enclose the description, end the banner. Parameter declarations appear immediately thereafter, followed by the curly-bracketed function code.

No explicit function return type is specific if the default **int** applies. Similarly, **int** formal parameters are not declared.

The function's statements are indented as follows: Each level of statement logic indentation is one full 8-column tab, and each continued line is indented under its parent line by 4 spaces (a half tab). Each **case** of a **switch ()** statement is indented 4 spaces from the **switch** on a separate line, and each case clause gets a full tab indentation.

Blank lines separate **return**, **break**, and **continue** statements from succeeding lines in functions and loops, except when the next line contains only a closing curly brace. In this case, the nearly blank line suffices to set off the early departure from normal processing. Blank lines also separate data structure declarations from the algorithmic code and appear in other places where they seem to bring better clarity.

Curly braces are always vertically aligned in function definitions, **struct** declarations, and nested control-logic, and usually also in data structure initializations.

All macro definitions have uppercase identifiers.

[7]Functions in the *ConJoin* program file appear in depth-first order.

7.2 Program Preamble and ANSI Header Files

The first block of code in the *ConJoin* program defines the program **VERSION**, declares the copyright notice string array, and invokes ANSI standard include-files. The array declaration at this point is not in the normal order of style described in Section 7.1; placed here, it serves both to display the copyright information that is required by JPL in its externally released software and provide a banner for announcing the program, when executed.

```
#define VERSION "(08-Apr-1992)" /*                              (ConJoin.c)*/
char copyrightnotice[14][76] =
{  "###########################################################################",
   "#                                                                         #",
   "#        Copyright (C) 1992, California Institute of Technology           #",
   "#      All rights reserved.  U. S. Government sponsorship under NASA      #",
   "#                   Contract NAS7-918 is acknowledged.                    #",
   "#                                                                         #",
   "#                         Robert C. Tausworthe                            #",
   "#                       Jet Propulsion Laboratory                         #",
   "#                          4800 Oak Grove Drive                           #",
   "#                       Pasadena, CA 91109-8099                           #",
   "#                                                                         #",
   "#                                                                         #",
   "###########################################################################",
   ""
};
/*
                              ANSI STANDARD HEADER FILES                  */

#include <ctype.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
```

Note, in this prototype form, the **VERSION** is identified by a date, rather than a release number. This date is automatically supplied by my (customized) text editor whenever I alter the file. The new date replaces whatever appears within the first set of parentheses in the file. The second parenthetical element is the name of the file, manually placed there for my convenience.

The header **ctype.h** is included because it contains function prototypes and/or macros for **isspace()** and **isprint()**, which are accessed by the functions **fgetstr()** (Section 7.26) and **isdigit()**, the latter of which is used by the **access()** function (Section 7.20).

The **string.h** header appears because the function prototypes for **strcat()**, **strchr()**, **strcmp()**, **strcpy()**, **strlen()**, and **strstr()** are accessed in various parts of the program.

The **stdio.h** header contains function prototypes and definitions of familiar input and output entities, such as **fclose()**, **getpos()**, **fgets()**, **FILE**,

fopen(), **fsetpos()**, **perror()**, **printf()**, and **stderr**, scattered throughout the program.

The **time.h** header provides definitions for the **tm** structure and **time_t** data type, and function prototypes for **asctime()**, **localtime()** and **time()**, all of which appear in the **timestamp()** function (Section 7.15).

The **stddef.h** header file is not used, because the only useful element in it was **NULL**; however, **NIL** is used instead (defined in Section 7.3 below).

7.3 Definitions, Defaults, and Macros

7.3.1 Synonyms

ConJoin defines a number of data types, storage classes, constants, and operators.

```
typedef int             BOOL;
typedef unsigned char   CHAR;
typedef unsigned char * STRING;

#define GLOBAL          extern
#define LOCAL           static

#define NIL             ((void *) 0)
#define NUL             0

#define FALSE           0
#define TRUE            1

#define AND             &&
#define IS              ==
#define NOT             !
#define MOD             %
#define OR              ||
```

I adopted synonyms for the storage type macros **extern** and **static**, viz., GLOBAL and LOCAL, for some now-forgotten, but probably cosmetic reason, many years ago. GLOBAL data and functions are accessible by all program file elements, while LOCAL entities are accessible only within the environment in which they are defined.

The null values NIL and NUL were defined for distinguishability among null-pointer and null-integer names. IS was defined because I was continually getting in trouble using = where == should have been (a carryover from programming in languages with no distinction between assignment and equality operators), and the other operators followed for similar reasons.

7.3.2 Manifest Constants

The manifest constants in *ConJoin* are

```
#ifndef FILENAME_MAX
#define FILENAME_MAX      50
#endif
#define MAX_CONTEXT       10
#define MAX_LINE         135
#define MAX_Q            100
#define NUMBER_OF_PATHS   20
#define PAGE_WIDTH        75
```

FILENAME_MAX is normally defined in **stdio.h**, as required by the ANSI standard. However, it does not appear there in the Microsoft C 5.1 used to develop the program, so I have defined it conditionally here. **MAX_LINE** is the maximum assumed length of a text line in a *ConJoin* source file. **MAX_Q** is the maximum number of lines that may be queued, and thus the maximum negative offset for lines transcribed by the **%access** directive. The **NUMBER_OF_PATHS** is the maximum number of **%path** directives that may appear in a *ConJoin* program. The **PAGE_WIDTH** value is used by **timestamp()** in writing the warning banner on the target file being produced (Section 7.15).

7.3.3 Directives

Since *ConJoin* directives are recognized and passed as comments to the target processor (TEX), it is necessary for *ConJoin* to recognize the comment syntactic conventions of the target processor. *ConJoin* recognizes directives as specially formatted strings appearing between **open** and **close** markers, whose values are defined by

```
#define OPEN_COMMENT    "%"
#define CLOSE_COMMENT   ""
#define COMMENT_LENGTH  10
```

These values are among the few in *ConJoin* that are not alterable by run time directives options. They limit *ConJoin* operations to TEX and its dialects. Other target processor comment delimiters can be accommodated only by recompilation.

Next are defined the *ConJoin* directive signal strings. When these appear, the actions described in Section 3 take place.

```
#define ACCESS_SIGNAL   OPEN_COMMENT "access"
#define BREAK_SIGNAL    OPEN_COMMENT "break"
#define COLUMN_SIGNAL   OPEN_COMMENT "column"
#define COUNT_SIGNAL    OPEN_COMMENT "count"
#define GARBAGE_SIGNAL  OPEN_COMMENT "garbage"
#define PATH_SIGNAL     OPEN_COMMENT "path"
#define POSTFIX_SIGNAL  OPEN_COMMENT "postfix"
#define PREFIX_SIGNAL   OPEN_COMMENT "prefix"
#define RANGE_SIGNAL    OPEN_COMMENT "range"
#define SHOW_SIGNAL     OPEN_COMMENT "show"
#define SIZE_SIGNAL     OPEN_COMMENT "size"
#define TAB_SIGNAL      OPEN_COMMENT "tabs"
```

7.3.4 Default Parameters

Certain *ConJoin* parameters can be altered by directives; others may be changed only by recompilation. Values for most of these are discussed in Section 3. Defaults are

```
#define BREAK_DEFAULT    ","
#define COLUMN_DEFAULT   0
#define COUNT_DEFAULT    "#"
```

```
#define GARBAGE_DEFAULT  '#'
#define OLD_TYPE_DEFAULT ".T_X"
#define POSTFIX_DEFAULT  "\\end" "{verbatim}}"
#define POSTVERBATIM     "|\n\n"
#define PREFIX_DEFAULT   "{\\footnotesize \\begin{verbatim}"
#define PREVERBATIM      "\\noindent \\verb|"
#define RANGE_DEFAULT    "-"
#define SHOW_DEFAULT     FALSE
#define TAB_DEFAULT      8
#define TGT_TYPE_DEFAULT ".TeX"
```

PREVERBATIM and POSTVERBATIM values are not discussed in Section 3, but they occur in response to the %show directive to provide descriptive markup tagging for displaying commands in the target document. PREVERBATIM contains \noindent to make the directive print at the left margin, and POSTVERBATIM contains \n\n to place the ensuing text onto a new line.

Error message strings are defined LOCALly within the error_message() function (Section 7.10). Access to these is made by indices defined globally:

```
#define BGN_MATCH_ERR   0
#define BREAK_ERR       1
#define CMD_LINE_ERR    2
#define END_MATCH_ERR   3
#define IO_SAME_ERR     4
#define MEMORY_ERR      5
#define NO_ACCESS_ERR   6
#define NO_COPY_ERR     7
#define RANGE_ERR       8
#define SIZE_ERR        9
```

See the narrative for the error_message() function, Section 7.10, for a discussion of why this potentially fragile approach to error messages was taken.

7.3.5 Macro Function

The final definition is a macro for reading a message (msg) from the size-file stream, removing trailing white space (including newline), and saving it in malloc-ated memory.

```
#define FGETSIZE(s)      strdup(strtrim(fgets(s, MAX_LINE, size_stream)))
```

7.4 Function Prototypes for the TOP-C Library

Earlier it was mentioned that several functions from the TOP-C library were accessed. *ConJo*in provides function prototypes for these,

```
GLOBAL STRING          stratrim(STRING);
GLOBAL STRING          strdup(STRING);
GLOBAL STRING          strfnb(STRING);
GLOBAL STRING          strinsert(STRING, STRING);
GLOBAL STRING          strlwr(STRING);
GLOBAL STRING          strnset(STRING, int, int);
GLOBAL STRING          strtcpy(STRING, STRING, int);
GLOBAL STRING          strtrim(STRING);
```

Source listings for these functions appear in Section 8.

7.5 Local Data Structures

Despite the "global data considered dangerous" caveat of structured programming purists (which I consider myself to be, at least in spirit [25]), a few data structures accessible via the overall environment were considered necessary. These appear in a number of functions where accessing them as formal parameters would be awkward, unsightly, and distracting. The data are

```
LOCAL long     access_lines;
LOCAL CHAR     close[COMMENT_LENGTH]        = CLOSE_COMMENT;
LOCAL int      column                       = COLUMN_DEFAULT;
LOCAL CHAR     ConJoin_file[FILENAME_MAX] = "";
LOCAL long     ConJoin_lines;
LOCAL FILE *   ConJoin_stream               = NIL;
LOCAL STRING   countsignal                  = COUNT_DEFAULT;
LOCAL BOOL     credits                      = TRUE;
LOCAL CHAR     garbage                      = GARBAGE_DEFAULT;
LOCAL STRING   last_acc_lines               = NIL;
LOCAL STRING   last_CJn_lines               = NIL;
LOCAL STRING   last_tgt_lines               = NIL;
LOCAL STRING   last_use_lines               = NIL;
LOCAL CHAR     mark[10]                     = BREAK_DEFAULT;
LOCAL CHAR     open[COMMENT_LENGTH]         = OPEN_COMMENT;
LOCAL STRING   path_list[NUMBER_OF_PATHS];
LOCAL int      path_list_size               = 0;
LOCAL CHAR     postfix[MAX_LINE]            = POSTFIX_DEFAULT;
LOCAL CHAR     postverbatim[30]             = POSTVERBATIM;
LOCAL CHAR     prefix[MAX_LINE]             = PREFIX_DEFAULT;
LOCAL CHAR     preverbatim[30]              = PREVERBATIM;
LOCAL CHAR     range[10]                    = RANGE_DEFAULT;
LOCAL BOOL     show                         = SHOW_DEFAULT;
LOCAL CHAR     size_file[FILENAME_MAX]      = "";
LOCAL CHAR     spaces[MAX_LINE]             = "";
LOCAL int      tabwidth                     = TAB_DEFAULT;
LOCAL CHAR     tgt_file[FILENAME_MAX]       = "";
LOCAL long     tgt_lines;
LOCAL FILE *   tgt_stream                   = NIL;
```

Data items that are not initialized above are set before use in the program. Many of the variables initialized here are *ConJoin* directive defaults: **column**, **count_signal**, **garbage**, **mark** (used instead of **break**, which is a C reserved word), **postfix**, **prefix**, **range**, **show**, and **tab_width**. Default values were discussed in Section 7.3. Path specifications, named in **path** directives, will be entered into the **path_list** array, whose size, initially zero, is maintained in **path_list_size**.

The **ConJoin_file** string will contain the source file name obtained from the user-entered command line, and **ConJoin_stream** will designate the corresponding **FILE *** stream. The number of lines read from the source is counted by the variable **ConJoin_lines**; in the event this file is very large, the type has been made **long**.

The target file name is held in the **tgt_file** string, and **tgt_stream** is the corresponding **FILE *** stream, when opened. The number of lines written into the target file by **%access**-es is **access_lines**, and the total number of lines written to the target file is **tgt_lines**, both **long**. All three numbers of lines are written to the file named by **size_file**; these numbers will be read into the variables **last_CJn_lines**, **last_acc_lines**, and **last_tgt_lines** when *ConJ*oin next processes the same source file.

The "announce" option, enabled by an **-a** entry on the command line, sets the **BOOL** variable **credits FALSE**, thereby suppressing printout of the program copyright notice, name, and version number in the **announce()** function of Section 7.9.

The two string variables **preverbatim** and **postverbatim** are unchanged once initialized in the program, so macro constants could replace them in the code (see **ConJoin_files()** in Section 7.17). Since these are references to LaTeX-dependent strings, they are candidates for **initialization()**-alteration in future versions, should that appear beneficial. If this need ever arises, it is an easy matter to replace the variables with macros.

The same is true of the variables **open** and **close**, which contain strings that open and close comments in the target-processor language.

The variable **spaces** is a string used by **ConJoin_files()**, **putline()**, and **timestamp()** functions for spacing lines of output to the target file.

7.6 The main() Program

The `main()` function of the *ConJoin* program is fairly short,

```
/****************************************************************************/

main(argc, argv)        /* Process a ConJoin file to create a target file.
                           Return a FALSE value if no failure occurs, or
                           TRUE or other nonzero value if a failure was
                           encountered.
/*------------------------------------------------------------------------*/
STRING argv[ ];
{
        BOOL    failure;
        FILE    *size_stream;
        int     i;

        initialization(argc, argv); /* terminates if no source file named     */
        open_io_files();            /* terminates on failure in opening files */
        access_lines = ConJoin_lines = tgt_lines = 0;
        timestamp();
        failure = ConJoin_files();
        failure |= fclose(ConJoin_stream) | fclose(tgt_stream);
        for (i = 0; i < path_list_size; i++)
                free(path_list[i]);
        free(last_acc_lines);
        free(last_CJn_lines);
        free(last_tgt_lines);
        free(last_use_lines);
        printf("Processed:\n%10ld %s source lines\n%10ld accessed lines\n"
            "%10ld %s total lines written\n", ConJoin_lines, ConJoin_file,
            access_lines, tgt_lines, tgt_file);
        if (size_stream = fopen(size_file, "w"))
        {       fprintf(size_stream, "%ld\n%ld\n%ld\n%.3f\n", ConJoin_lines,
                    access_lines, tgt_lines,
                    (double) access_lines / (double) tgt_lines);
                failure |= fclose(size_stream);
        }
        return failure;
}
```

The usual `main()` command-line arguments **`argc`** and **`argv`** are passed directly to `initialization()`, from which a mandatory `ConJoin_file` name, an optional target name, and an optional switch to disable the credits announcement are extracted. If a target is not named on the command line, a default name is made from the `ConJoin_file` by replacing the source file type with a default type (currently, `.TeX`). If no source file is named, or if the source and target files are the same, `initialization()` prints a usage message, then terminates the program with an exit value of `TRUE`. All error terminations return a nonzero value to the operating system in case *ConJoin* has been invoked from a script (or batch) file whose further processing may be affected.

Both source and target files are opened; if this is not possible, the cause of the failure and the usage message appear, then *ConJoin* terminates as above. If a target file of that name already exists, it is renamed using an `OLD_TYPE_DEFAULT` (see default values in Section 7.3, above).

A header is written by the `timestamp()` function to the target file before any processing takes place. This banner stamps the file with the time and date it was created, and also records the target file name. The name of the source file is also written, along with an instruction not to edit the target file, but to make changes in the *ConJoin* source instead. (The reason for this is that editing the target file will not survive the next *ConJoin*-ing of the same source file.)

The main work of the program occurs in `ConJoin_files()`. This function reads lines from the source file and examines whether a *ConJoin* directive appears. If so, the action described in Section 3 occurs; if not, the line is copied to the target file intact.

If an error occurs, either in *ConJoin*-ing or closing files, the nonzero error value is set for termination, `malloc`-ated strings from the `%path` directive and elsewhere are freed (this may be done automatically by the operating system upon program termination, and could be redundant). This is followed by a terminating summary message and by saving the numbers of source, access, and target lines.

7.7 The initialization() Function

```
/**************************************************************************/
        void
initialization(argc, argv)
        /* Process command line file names and options, and retrieve
           size_file statistics.
/*-----------------------------------------------------------------------*/
STRING argv[ ];
{
        CHAR    msg[MAX_LINE];
        STRING  s;
        FILE    *size_stream;

        command_line(argc, argv, msg);
        announce();
        if (*msg)
        {       usage();
                error_message(CMD_LINE_ERR, msg, FALSE);
                exit(TRUE);
        }
        file_defaults();
        if (NOT strcmp(ConJoin_file, tgt_file))
        {       strcpy(msg, ConJoin_file);
                *ConJoin_file = NUL;
                usage();
                error_message(IO_SAME_ERR, msg, FALSE);
                exit(TRUE);
        }
        strcpy(strchr(s = strcpy(size_file, tgt_file), '.'), ".siz");
        if (size_stream = fopen(size_file, "r"))
        {       last_CJn_lines = FGETSIZE(msg);
                last_acc_lines = FGETSIZE(msg);
                last_tgt_lines = FGETSIZE(msg);
                last_use_lines = FGETSIZE(msg);
                fclose(size_stream);
        }
}
```

The **initialization()** function starts by passing the **argc** and **argv** program inputs to the **command_line()** processor, which proceeds through the **argv** strings looking for options and file names. Each unrecognized option is concatenated with any others and recorded in the **msg** string. (Caution: too many or long anomalous inputs may cause **msg** to exceed its allotted width, and may bomb the program in some implementations. If this appears to be a problem, future releases may adopt more robust handling of such conditions.) The first non-option string is assumed to be the **ConJoin_file**, the second, the **tgt_file**, and any others are errors, appended to the **msg** string.

Next, the program **announce()**-ment is made. If **credits** has been set **FALSE**, the copyright, program, and version printout are inhibited.

If a **msg** has been returned from initialization, a command-line error has occurred. **usage()** prints a short set of operational instructions, and the error message follows. The function returns a **TRUE** value to the operating system.

The `file_defaults()` function supplies a target file name and default file types if these were unspecified on the command line. If no `tgt_file` has been named, the `ConJoin_file` name is used, with the file type changed to the target type default. If a `tgt_file` is named, but no "." appears in the file name, the default type is again appended.

If the names of the `ConJoin_file` and the `tgt_file` are the same, the usage message and error are printed and the program again terminates with a returned value of **TRUE**. The `ConJoin_file` string is nulled so that no file line and column number appear in the error message.

Finally, the function ends by constructing the name of the `size_file`, and then retrieving the last-time values of source, accessed, and target lines from it for use by the `%size` directive.

7.8 The command_line() Function

```
/************************************************************************/
        void
command_line(argc, argv, msg)
        /* Process information on the command line: extract ConJoin_file
           and tgt_file names, and option -a, when present.  Return with
           msg set to error conditions.
/*----------------------------------------------------------------------*/
STRING argv[ ], msg;
{
        int     i;
        STRING  s;

        *msg = NUL;
        for (i = 1; i < argc; i++)
        {       s = strlwr(argv[i]);
                if (*s IS '-' OR *s IS '/')
                {       switch (*++s)
                        {   case 'a':
                                credits = FALSE;
                                break;
                            default:
                                sprintf(msg + strlen(msg), "Unknown option: "
                                    "%s.\n", argv[i]);
                        }
                        *argv[i] = NUL;
                }
                else if (NOT *ConJoin_file)
                        strcpy(ConJoin_file, s);
                else if (NOT *tgt_file)
                        strcpy(tgt_file, s);
                else
                        sprintf(msg + strlen(msg), "Unknown command: %s.\n", s);
        }
        if (NOT *ConJoin_file)
                strcat(msg, "No source file named.\n");
}
```

This function first **NIL**s the names of source and target files and the return message, and then sequences through the command line arguments (excluding the 0th, which is the program path), one by one. Any argument beginning with - or / is deemed an option, and only **-a** is presently acceptable; its appearance turns off the program, copyright notice, author, and version **credits**. Any other attempted option input concatenates an **Unknown option:** *string* onto **msg**.

The first command line argument not deemed an option is taken to be the **ConJoin_file** name, and the second, that of the **tgt_file**. Any others append **Unknown command:** *string* to **msg**.

The use of **sprintf()** to concatenate error strings is a simple way to produce the formatted **msg** involving the offense, the offending element, and a newline.

If no source file appears on the command line, this fact is appended to the error **msg**.

7.9 The announce() Function

If the `credits` switch is still intact (i.e., TRUE), then the copyright notice, author, program name, and version are printed; if not, these items are omitted.

```
/*************************************************************************/
        void
announce()                 /* Announce program, copyright, and author.

/*----------------------------------------------------------------------*/
{
        int     i;

        if (credits)
        {       for (i = 0; *copyrightnotice[i]; i++)
                        printf("%s\n", copyrightnotice[i]);
                printf("\n\t\t\t   ConJoin Program"
                       "\n\t\t\t    %s\n\n", VERSION);
        }
}
```

7.10 The error_message() Function

```
/****************************************************************************/
        void
error_message(n, s, f)  /* Write error message n augmented with string s
                           to stdout, indicating the current line in the
                           source file.  Repeat the message on the target
                           file if f is TRUE.
/*------------------------------------------------------------------------*/
STRING s;
{
        LOCAL STRING  errmsg[ ] =
        {       "Beginning match string not found: ",
                "Break string is invalid: ",
                "Command line error: ",
                "End-match string not found: ",
                "Input and output files may not be the same: ",
                "Memory insufficient for queue.",
                "No access file found: ",
                "No lines copied from accessed file.",
                "Range separator missing.",
                "Size command case invalid: "
        };

        if (*ConJoin_file)
                printf("\a%s, %ld 1: %s%s\n",
                    ConJoin_file, ConJoin_lines, errmsg[n], s);
        else
                printf("%s%s\n", errmsg[n], s);
        if (f)
                fprintf(tgt_stream, "***ERROR*** %s%s\n", errmsg[n], s);
}
```

The LOCALized declaration of error messages accessed via globally defined macro indexes is admittedly fragile and prone to unreliability when later changes are made in the program's error handling. It would have been more reliable to pass the error message directly, as a string argument, rather than as a numeric index. Each error message, in either case, appears only once in the program, so there is no storage advantage whether localized or dispersed. I tried it both ways. I think that the program is more readable with all the messages in one place. If, in future maintenance, this decision proves faulty, redistribution of error messages will be considered.

The **error_message()** function merely prints the selected error message to **stdout**, and, if **f** has been set **TRUE**, also to the **tgt_stream** in slightly altered form.

7.11 The usage() Function

```
/****************************************************************************/
        void
usage()                 /* Print a message on usage syntax of ConJoin.

/*------------------------------------------------------------------------*/
{
        printf("Usage:  ConJoin <ConJoin source> <target file> "
            "[<options>]\n\n"
            "\tSource file type default is .CJn\n"
            "\tTarget file type default is %s\n\n"
            "Options:\n"
            "\t-a      Do not announce the program.\n", TGT_TYPE_DEFAULT);
}
```

This function is invoked whenever an abortive usage error has occurred. The message reminds the user of the syntax that is required, the defaults that apply to that syntax, and the processing options that are available. It terminates with a TRUE value returned to the operating system.

7.12 The file_defaults() Function

```
/*************************************************************************/
        void
file_defaults()         /* Supply ConJoin file type .CJn if missing, and
                           supply missing parts of tgt_file, if any.
/*-----------------------------------------------------------------------*/
{
        STRING  s;

        if (NOT (s = strchr(ConJoin_file, '.')))
                strcat(ConJoin_file, ".CJn");
        if (NOT *tgt_file)
        {       strtcpy(tgt_file, ConJoin_file,
                    strchr(ConJoin_file, '.') - ConJoin_file);
        }
        if (NOT (s = strchr(tgt_file, '.')))
                strcat(tgt_file, TGT_TYPE_DEFAULT);
}
```

Note that the **ConJoin_file** will always have a file type; if one is not provided by user input, it is supplied in the first **if** clause. If no **tgt_file** has been named, the **ConJoin_file** name is used, up to the "." (a "." is guaranteed to appear by the first step above). The length of text copied excludes copying of the dot.

If the **tgt_file** bears no ".", the **TGT_TYPE_DEFAULT** is applied. Thus, the target file also always has an explicit file type.

7.13 The open_io_files() Function

```
/****************************************************************************/
        void
open_io_files()         /* Open ConJoin_file and tgt_file into ConJoin_stream
                           and tgt_stream.  Rename old tgt_file, if any,
                           with OLD_TYPE_DEFAULT.  Terminate with an error
                           message via file_open() if files cannot be opened.
/*------------------------------------------------------------------------*/
{
        CHAR    tgt_bak[FILENAME_MAX];
        FILE *  f;
        STRING  s;

        ConJoin_stream = file_open(ConJoin_file, "r");
        if (f = fopen(tgt_file, "r"))
        {       fclose(f);
                s = strchr(strcpy(tgt_bak, tgt_file), '.');
                strcpy(s, OLD_TYPE_DEFAULT);
                remove(tgt_bak);
                rename(tgt_file, tgt_bak);
        }
        tgt_stream = file_open(tgt_file, "w");
}
```

The first **fopen()** finds the *ConJoin* source file for reading. If the file cannot be found, or otherwise cannot be opened, the program terminates at this point with an error message and a **TRUE** value returned to the operating system.

The **if** clause checks to see whether the file named by **tgt_file** currently exists by opening the file for reading. If it does, a backup file name is created using the name up to the "." followed by the **OLD_FILE_TYPE**. Then the current file is renamed to this **tgt_bak**.

The final **fopen()** opens the target file for writing, with the same termination consequences as above if the target cannot be opened.

7.14 The file_open() Function

This function is equivalent to **fopen()**, except that if the file cannot be opened, an error message is printed with the reason followed by program termination. The function is only invoked where abortive action is desired in response to an error. The name is nulled prior to calling **error_message** to suppress the file name and location portions of the printout.

```
/*************************************************************************/
	FILE *
file_open(name, use)
	/* Open the name file for given use, and return resulting stream.
	   If file cannot be opened, print reason, and abort processing.
/*-----------------------------------------------------------------------*/
STRING name, use;
{
	CHAR	s[MAX_LINE];
	FILE *	stream;

	if (NOT (stream = fopen(name, use)))
	{	strcpy(s, name);
		*name = NUL;
		strncat(strcat(s, ": "), strerror(errno), MAX_LINE - 1);
		error_message(CMD_LINE_ERR, s, FALSE);
		exit(TRUE);
	}
	return stream;
}
```

7.15 The timestamp() Function

The purpose of the **timestamp()** function is to annotate the target file being written with informative and warning commentary. The information consists of the target file name (**tgt_file**), the current time (accessed via ANSI standard functions **time()** and **localtime()**), and the name of the **ConJoin_file** that created it. The warning is simply an exhortation not to revise the file. Nothing dire happens if this file is altered, but the user probably meant for the changes to have been made in the original source file. When *ConJoin*-ed again, changes in the target file will be lost.

```
/***************************************************************************/
        void
timestamp()     /* Write the tgt_file name and a time-stamped header with
                   a revision warning onto the tgt_stream.
/*-------------------------------------------------------------------------*/
{
        time_t          clock;
        CHAR            atime[26],
                        bar[MAX_LINE],
                        blanks[MAX_LINE];
        STRING          sp, text;
        struct tm *     t;
        int             n;

        n = strlen(open) + strlen(close);
        strnset(bar, '=', PAGE_WIDTH - n);
        strnset(blanks, ' ', PAGE_WIDTH - n - 2);
        time(&clock);
        t = localtime(&clock);
        stratrim(strcpy(atime, asctime(t)));
        sp = right_fill(atime, strlen(tgt_file) + n + 5);
        fprintf(tgt_stream, "%s (%s)%s(%s)%s\n", open, atime, sp,
            tgt_file, close);
        fprintf(tgt_stream, "%s%s%s\n", open, bar, close);
        text = "%s|        This file was ConJoin-ed from input file %s.%s|%s\n";
        sp = right_fill(text, strlen(ConJoin_file) - n - 7);
        fprintf(tgt_stream, text, open, ConJoin_file, sp, close);
        fprintf(tgt_stream, "%s|%s|%s\n", open, blanks, close);
        text = "%s|                  DO NOT REVISE THIS FILE.%s|%s\n";
        fprintf(tgt_stream, text, open, right_fill(text, n - 7), close);
        fprintf(tgt_stream, "%s|%s|%s\n", open, blanks, close);
        text = "%s|        To make revisions, modify the original file.%s|%s\n";
        fprintf(tgt_stream, text, open, right_fill(text, n - 7), close);
        fprintf(tgt_stream, "%s%s%s\n", open, bar, close);
        tgt_lines += 8;
}
```

The only tricky part of this function is in correctly creating the flush-right time-stamp box. The function **strnset()** from TOP-C creates **bars** for the top and bottom of the box, while **right_fill()** provides the correct number of blanks for formatting. Even though the global variable **spaces** is not referenced within **timestamp()**, it is changed as a side-effect of **right_fill()** (see the discussion in Section 7.16, below).

The reason for using **stratrim()** before copying **asctime()** to **atime** is that the string returned by **asctime()** is terminated in a newline character, and it is necessary to remove the trailing white space for proper formatting.

The **tgt_lines** variable is finally augmented by the number of lines written onto the target file by this function.

As an example, the header written by **timestamp()** on the target file comprising the body of this report is

```
% (Fri Jul 31 11:20:39 1992)                                  (cj_body.TeX)
%==========================================================================
%|          This file was ConJoin-ed from input file cj_body.CJn.        |
%|                                                                        |
%|                    DO NOT REVISE THIS FILE.                            |
%|                                                                        |
%|          To make revisions, modify the original file.                  |
%==========================================================================
```

7.16 The right_fill() Function

```
/***************************************************************************/
        STRING
right_fill(s, n)        /* Generate spaces as a blank string of length
                           PAGE_WIDTH - n.  Return spaces.
/*-------------------------------------------------------------------------*/
STRING  s;
{
        return strnset(spaces, ' ', PAGE_WIDTH - strlen(s) - n);
}
```

This simple function perhaps exemplifies the practice of undue parsimony in programming: the use of an existing, idle global data structure by a function it was not initially intended to serve. Throughout the remainder of the program, **spaces** carries the indentation prefix for lines copied from accessed files. But here, before the scanning of the source even begins, it is free for other duty. Luckily, its name fits both usages. This general practice can lead to very fragile and hard-to-read code if overdone. Even though this module is only invoked from a single function of the program, and even though it is very tiny, the program modularity is less than optimum.

This function should be noted as a possible target for later perfective maintenance.

7.17 The ConJoin_files() Function

This function processes the *ConJoin* source file to produce the target file via the directives described earlier, in Section 3. The algorithm is straightforward:

```
/***************************************************************************/

ConJoin_files()          /* Process the ConJoin source file and create the
                            expanded target file.  Return a nonzero value
                            if an error occurs, or a 0 value if none.  Count
                            ConJoin_lines, access_lines, and tgt_lines.
/*-------------------------------------------------------------------------*/
{
        BOOL    error;
        CHAR    hold[MAX_LINE],
                line[MAX_LINE];
        STRING  extract;

        error = FALSE;
        *spaces = NUL;
        for (ConJoin_lines = 0; fgets(line, MAX_LINE, ConJoin_stream); )
        {       ConJoin_lines++;
                tgt_lines++;
                if (NOT *strfnb(line))
                {       fprintf(tgt_stream, "%s", line);
                        continue;
                }
                if (strstr(line, open))
                {       if (extract = strstr(line, ACCESS_SIGNAL))
                                strcpy(hold, line);
                        error |= directive(extract, line);
                }
                else
                        extract = NIL;
                fprintf(tgt_stream, "%s", line);
                if (extract)
                        error |= access(hold + (extract - line) +
                            strlen(ACCESS_SIGNAL));
        }
        return error;
}
```

The **error** switch is set **FALSE** and **spaces** is **NUL**-led in preparation for processing. The loop iteratively reads in a string **line** until an end-of-file or reading error occurs. Each line read in augments the number of **ConJoin_lines** and **tgt_lines**. All source lines copy into the target file in one form or another. If a line is blank, it is immediately written to the target file, and processing continues.

Each **line** is prescanned to see if it contains a *ConJoin* directive, as indicated by the appearance of a comment opener. If one exists, then the **line** is checked for the appearance of an **ACCESS_SIGNAL**, whose location is saved in the variable **extract**. If **line** is recognized as an **%access** it is saved in the **hold** buffer.

Thereafter, the **line** is processed as a possible directive. A non-**NIL** **extract** value sent to **directive()** is a signal that the directive is an **%access** and is only to be prepared for showing, if the **show** mode is in effect. The remainder of the processing of an **%access** directive takes place after the **line** has been written on the target.

If no comment opener was detected, **extract** is set to **NIL** so no **access()** action takes place later.

The **line** either prints as it was when read in, or as altered by **directive()**. Only **%size** and **%show** directives cause alterations to the **line**; **%size** inserts a numeric value into the **line**, and **%show** surrounds the **line** with **preverbatim** and **postverbatim** target processor commands when turned **on**.

If **extract** is non-**NIL**, the **hold** line is processed as an **%access** directive (the **line** will have been corrupted in the **show** mode). The position after the **%access** signal in **hold** is computed from its location in the **line** prior to **directive()** processing.

Processing **%access** directives is delayed with respect to processing of other directives so that transcribing the extracted text can follow the copying of the directive itself onto the target file.

On completion of scanning the source file, the function returns the compound results of error detection.

7.18 The directive() Function

Because of its length, this function will be described in increments. Overall, the structure is comprised of three communicating parts:

- declaration and initialization
- directive action
- preparation for showing

The first action is to preserve the incoming **text** by copying it into the **line** string, which is processed instead. However, **text** will also be processed if it contains a **%size** directive or if the **show** condition is **TRUE**. The **text** string is later written onto the target file by the invoking function.

```
/***************************************************************************/
        BOOL
directive(extract, text)
        /* Process ConJoin directives that may appear in the text string.
           If extract is non-NIL, the text contains an access directive
           that may only need to be prepared for show-ing.  In case of
           %size, write the appropriate values into text.  Insert
           preverbatim and postverbatim into text if show is, or has just
           turned, TRUE.  Return TRUE if a bad %size case appears; FALSE
           otherwise.
/*-----------------------------------------------------------------------*/
STRING extract, text;
{
        STRING  s,
                t;
        CHAR    g,
                line[MAX_LINE];
        BOOL    change_show;

        strcpy(line, text);
```

The next, and largest, segment of the function is a 12-way **if...else if ...else if** ...directive-selection structure. If **extract** is non-**NIL**, a delayed **%access** directive is in effect, so no action takes place in this step. The **t** pointer is set to the location of the beginning of the directive in the **line**.

```
        if (extract)
                t = line + (extract - text);
```

Many of the other directive actions simply record parameter values:

```
        else if (t = strstr(line, BREAK_SIGNAL))
                strext(mark, t + strlen(BREAK_SIGNAL));

        else if (t = strstr(line, COUNT_SIGNAL))
                strext(countsignal, t + strlen(COUNT_SIGNAL));
```

```
else if (t = strstr(line, POSTFIX_SIGNAL))
        strext(postfix, t + strlen(POSTFIX_SIGNAL));
else if (t = strstr(line, PREFIX_SIGNAL))
        strext(prefix, t + strlen(PREFIX_SIGNAL));
else if (t = strstr(line, RANGE_SIGNAL))
        strext(range, t + strlen(RANGE_SIGNAL));

else if (t = strstr(line, GARBAGE_SIGNAL))
{       if (g = *strfnb(t + strlen(GARBAGE_SIGNAL)))
                garbage = g;
}

else if (t = strstr(line, TAB_SIGNAL))
        tabwidth = atoi(t + strlen(TAB_SIGNAL));
```

In each of these, the remainder of the directive after the detected **SIGNAL** determines the new value of the parameter (white space suppressed). The parameters affected are **mark**, **countsignal**, **garbage**, **prefix**, **postfix**, **range**, and **tabwidth**.

A similar action takes place with **path_list** items, except each path extracted from the directive is saved in **malloc**-ated memory by **strdup()** of Section 8.2.

```
else if (t = strstr(line, PATH_SIGNAL))
{       path_list[path_list_size++] =
            strdup(strext(t, t + strlen(PATH_SIGNAL)));
}
```

If a **%column** directive appears with a positive value, the **spaces** string is filled with an equal number of blank characters; otherwise **spaces** is nulled.

```
else if (t = strstr(line, COLUMN_SIGNAL))
{       column = atoi(t + strlen(COLUMN_SIGNAL));
        if (column > 0)
        {       strnset(spaces, ' ', column);
                column = 0;
        }
        else
                *spaces = NUL;
}
```

A **%show** directive line is examined for **on** or **off** alternatives.

```
else if (t = strstr(line, SHOW_SIGNAL))
{       s = strlwr(stratrim(t + strlen(SHOW_SIGNAL)));
        if (NOT strcmp(s, "on"))
                change_show = TRUE + TRUE;
        else if (NOT strcmp(s, "off"))
                change_show = TRUE + FALSE;
        if (change_show AND NOT show)
                show = change_show - TRUE;
}
```

Either **on** or **off** causes **change_show** to switch from 0 (**FALSE**) to 1 (**TRUE + FALSE**) if **show** goes **off**, or to 2 (**TRUE + TRUE**) if **show** goes **on**. Unrecognized **show** alternatives are ignored. If **%show** has called for action (**change_show** has been set to a nonzero value), and if **show** is currently **off**, then whatever action was called for takes place immediately, in case the directive was to go **on**.

The remaining directive, **%size**, causes the action

```
else if (t = strstr(line, SIZE_SIGNAL))
{       do
        {       *t = NUL;
                switch (*(t = strfnb(t + strlen(SIZE_SIGNAL))))
                {   case 'c':
                    case 'C':
                        s = last_CJn_lines;
                        break;
                    case 'a':
                    case 'A':
                        s = last_acc_lines;
                        break;
                    case 'r':
                    case 'R':
                        s = last_use_lines;
                        break;
                    case 't':
                    case 'T':
                        s = last_tgt_lines;
                        break;
                    default:
                        s = "0";
                        error_message(SIZE_ERR, text, FALSE);
                        return TRUE;
                }
                sprintf(text, "%s%s%s", line, s, ++t);
        } while (t = strstr(strcpy(line, text), SIZE_SIGNAL));
}
```

Nulling the character where the **SIZE_SIGNAL** was found truncates the **line** at that point, removing the **%size** directive. The character after the **SIZE_SIGNAL** causes the corresponding number of lines saved in the size history file to replace the **%size** directive in the reconstructed **text** by **sprintf()**. The **do** loop iterates until no further **%size** directive is detected on the line.

The final segment of the **directive()** function prepares the line for possible explicit display in the target document. It does this by inserting pre- and post-verbatim environment commands around the directive. String **t** will be non-**NIL** if a directive was detected.

```
if (t AND show)
{       strinsert(text + (t - line), preverbatim);
        if (*close AND (t = strstr(text, close)))
                strinsert(text, postverbatim);
        else
```

```
                        strcat(stratrim(text), postverbatim);
                if (change_show)
                        show = change_show - TRUE;
        }
        return FALSE;
}
```

If no directive was found in **text**, the function terminates with no action taken.

7.19 The strext() Function

The purpose of this function is to extract the substring up to the closing comment mark (if any) and copy it into another, trimmed of leading and trailing white space.

```
/**************************************************************************/
        STRING
strext(s, t)    /* Extract string t up to the close-comment string, if close
                   is not null, or to the end of t if null, into s. Remove
                   leading and trailing blanks from s and return s.
/*----------------------------------------------------------------------*/
STRING  s, t;
{
        STRING p;

        if (*close AND (p = strstr(t, close)))
                strtcpy(s, t, p - t);
        else
                strcpy(s, t);
        return stratrim(s);
}
```

7.20 The access() Function

```
/*************************************************************************/
        BOOL
access(buffer)          /* Process the text extraction operation specified
                           in the line buffer to the tgt_stream.  Return
                           FALSE if no error, TRUE if an error occurred.
/*---------------------------------------------------------------------*/
STRING  buffer;
{
        STRING  bgn_match[MAX_CONTEXT + 1],
                end_match[MAX_CONTEXT + 1];
        CHAR    module[FILENAME_MAX];
        BOOL    error;
        FILE *  modulestream;
        int     bgncount,
                bgnoffset,
                endcount,
                endoffset,
                i;

        strext(buffer, buffer);
        if (match_parameters(buffer, module, bgn_match, end_match, &bgncount,
            &bgnoffset, &endcount, &endoffset))
                return TRUE;

        if (NOT (modulestream = open_access(module)))
        {       error_message(NO_ACCESS_ERR, module, FALSE);
                return TRUE;
        }
        printf("%saccess %s%s\n", open, buffer, close);
        if (*prefix)
                fprintf(tgt_stream, "%s\n", prefix);
        error  = scan_to_bgn_match(modulestream, bgn_match, bgncount,
            bgnoffset);
        error |= copy_to_end_match(modulestream, end_match, endcount,
            endoffset);
        for (i = 0; bgn_match[i]; i++)
                free(bgn_match[i]);
        for (i = 0; end_match[i]; i++)
                free(end_match[i]);
        error |= fclose(modulestream);
        if (*postfix)
                fprintf(tgt_stream, "%s\n", postfix);
        return error;
}
```

The first step of the algorithm removes the trailing comment signal and leading and trailing white space from **buffer**. The second step extracts the name of the file to access and the search parameters; if no error is encountered, the algorithm proceeds.

The third step opens the module stream; if the extracted file name is not found in the current directory, each of the directories named in **%path** directives is searched. If the file is found and opened successfully, the next step lists the name and extracted parameters by using current settings of the **%access** separator strings.

Next begins the actual access: If there is a **prefix**, it is written into the target file. The access file is searched for the beginning match conditions described in Section 3.2, and thence copies until the ending match conditions have been met. Finally, the access file is closed, the **postfix** is written to the target file, and the function terminates, returning the indicator of any error condition that may have occurred.

Note: the error expression contained in the three statements near the end of the function should not be combined into a single statement because the order of evaluation of functions is unspecified in the ANSI standard.

7.21 The match_parameters() Function

```
/*************************************************************************/
          BOOL
match_parameters(buffer, module, bgn_match, end_match, bgncount, bgnoffset,
     endcount, endoffset)
          /* Extract access module name, and beginning and ending access
             conditions.  Return TRUE if an error is encountered, FALSE
             otherwise.
/*---------------------------------------------------------------------*/
STRING  buffer, bgn_match[ ], end_match[ ], module;
int     *bgncount, *bgnoffset, *endcount, *endoffset;
{
        CHAR    line[MAX_LINE];
        STRING  s,
                t;

        if (NOT (s = strstr(strcpy(line, buffer), mark)))
        {       error_message(BREAK_ERR, buffer, FALSE);
                return TRUE;
        }
        stratrim(strtcpy(module, line, s - line));
        if (NOT (*(t = strfnb(s + strlen(mark)))
            AND (s = strstr(t, range))))
        {       error_message(RANGE_ERR, buffer, FALSE);
                return TRUE;
        }
        *s = NUL;
        s += strlen(range);
        if (access_condition(t, bgn_match, bgncount, bgnoffset, 1))
                return TRUE;

        if (access_condition(s, end_match, endcount, endoffset, -1))
                return TRUE;

        return FALSE;
}
```

The first steps above extract the name of the file to access into **module** by locating the **%break mark** (signaling error if none is found) and copying that portion of the input buffer.

The next segment sets **t** to the starting-match condition, beginning just past the **mark** location and extending up to the **range** marker. (If no marker is present, the function terminates returning an error indication.) Putting **NUL** in place of the marker isolates the starting match specification string.

Note that string **s** has been positioned just past the **range** marker, and therefore points to the end-condition string. The **access_condition()** function invocations extract the context strings, context count, and offset values for the matches to be made in scanning for the beginning match and copying to the ending match.

7.22 The access_condition() Function

Access match conditions are extracted from the string **buffer** parameter. The context strings are copied into the **match** dynamic string array, but **count** and **offset** values are first set by locating these terms and removing them from the **buffer**. Removal merely requires seeking for **countsignal**, +, and -, then converting substrings to integer values, and finally inserting **NULs** to isolate the match-string portion of the condition.

```
/****************************************************************************/
        BOOL
access_condition(buffer, match, count, offset, init)
        /* Extract access condition from buffer. Set offset to init if
           no offset is parsed in the buffer.  Return TRUE if an error
           is encountered, FALSE otherwise.
/*--------------------------------------------------------------------------*/
STRING  buffer, match[ ];
int     *count, *offset;
{
        int     n;
        STRING  s, t;

        *offset = init;
        *count = 1;
        s = stratrim(buffer);
        if (t = strstr(s, countsignal))
        {       *count = atoi(s = t + strlen(countsignal));
                *t = NUL;
        }
        if ((t = strchr(s, '+')) OR (t = strchr(s, '-')))
        {       *offset = atoi(t);
                *t = NUL;
        }
        else if (isdigit(*buffer))
        {       *offset = atoi(buffer);
                *buffer = NUL;
        }
        s = buffer;
        for (n = 0; *s AND n < MAX_CONTEXT; n++)
        {       if (t = strstr(s, mark))
                {       *t = NUL;
                        t += strlen(mark);
                }
                match[n] = strdup(strtrim(s));
                s = strfnb(t AND *t ? t : s + strlen(s));
        }
        if (n > 0 AND match[n - 1] IS NIL)
                return TRUE;

        match[n] = NIL;
        return FALSE;
 }
```

7.23 The open_access() Function

The **open_access()** function attempts to open the **file** named as the parameter. Failing this, it attempts opening this file in each of the directories previously named in **%path** directives, until successful. Then, the open file stream is returned. If unsuccessful, a **NIL** value is returned to the calling procedure, **access()**, of Section 7.20.

```
/***************************************************************************/
        FILE *
open_access(file)        /* Open the specified file for reading.

/*-------------------------------------------------------------------------*/
STRING  file;
{
        int     i;
        CHAR    path[FILENAME_MAX];
        FILE *  stream;

        strcpy(path, file);
        for (i = 0; NOT (stream = fopen(path, "r")) AND i < path_list_size; i++)
                strcat(strcpy(path, path_list[i]), file);
        return stream;
}
```

7.24 The scan_to_bgn_match() Function

```
/****************************************************************************/
        BOOL
scan_to_bgn_match(modulestream, bgn_match, bgncount, bgnoffset)
        /* Scan the modulestream for the beginning match condition and
           return positioned to access the first line to be copied.
/*------------------------------------------------------------------------*/
FILE *  modulestream;
STRING  bgn_match[ ];
int     bgncount, bgnoffset;
{
        fpos_t  fp,
                fpqueue[MAX_Q];
        int     n,
                offset,
                qex;
        BOOL    error;
        long    qe;
        CHAR    text[MAX_LINE];

        n = error = offset = 0;
        qe     = -1;
        while (NOT fgetpos(modulestream, &fp) AND
            fgets(text, MAX_LINE, modulestream))
        {       qex = (int)(++qe MOD MAX_Q);
                fpqueue[qex] = fp;
                if (bgn_match[n])
                {       if (NOT strstr(text, bgn_match[n]))
                                        continue;

                        if (bgn_match[++n])
                                continue;

                        if (--bgncount <= 0)
                        {       if (bgnoffset <= 0)
                                {       qex = (int)((qe + bgnoffset) MOD MAX_Q);
                                        fsetpos(modulestream, &fpqueue[qex]);
                                        break;
                                }
                                else if (bgnoffset-- IS 1)
                                        break;
                        }
                        else
                                n = 0;
                }
                else if (NOT bgnoffset)
                {       fsetpos(modulestream, &fpqueue[qex]);
                        break;
                }
                else if (++offset >= bgnoffset)
                        break;
        }
        if (bgn_match[0] AND bgn_match[n])
        {       error_message(BGN_MATCH_ERR, bgn_match[n], TRUE);
```

```
                error = TRUE;
        }
        return error;
}
```

In this algorithm, **qb** and **qe** are the queue-beginning and queue-ending indices of lines read from the access file. If the beginning offset is negative, then when the beginning match and count conditions are reached, the beginning line is located from the queue. The lines themselves are not queued, but rather their positions in the file are stored in a file-position-queue, **fpqueue[ ]**. Since the queue length is limited to **MAX_Q** elements, it is necessary to treat the queue circularly, indexed by integers **qbx** and **qex**.

Until the beginning access condition is met, the file positions and text lines are read into **fp** and **text**, respectively. The position is saved in **fpqueue[qex]**, while **text** is examined for beginning match conditions.

One will note that the **copy_to_end_match()** function (Section 7.25) maintains a text queue, rather than a file position queue. That alternative was also considered here to make the two functions more similar. The speed trade-off between the two alternatives—i.e., constantly allocating, copying, and freeing strings into and from the queue vs. the one-time repositioning of the file—seemed moot. The use of the text queue, however, required that the copying function also needed access to the queue. Manipulation of the queue in copying seemed less straightforward and much different from this scanning function. Simplicity and similarity finally won out.

As long as the **bgn_match[n]** string is non-null, there is a match to be found; otherwise, the string matching condition is considered satisfied. Whenever a match is satisfied, its count is decremented, and the next occurrence of the **bgn_match** array is sought (at $\mathtt{n} = 0$).

Upon reaching the context and count conditions, if the beginning offset was zero or negative, the queued location of the line read **bgnoffset** lines earlier is used to reposition the input file to next read at the specified line. The loop breaks at this point and the function terminates.

If the beginning offset value is positive, then **bgnoffset** is decremented to count (and thereby skip) the line. If the offset (before the decrement) is one, then the loop breaks and the function terminates, as the end matching condition has been fulfilled.

If the function began without a context to be matched and no offset, then the file is reset to the beginning, and the function terminates. If there was a beginning offset only, an **offset** variable counts up until the specified offset is reached before the loop is broken.

If, after the loop terminates, if **bgn_match[0]** and **bgn_match[n]** both are un-nulled, this is an indication that a match was to be found, but was not. An error message results in this case.

7.25 The copy_to_end_match() Function

The `copy_to_end_match()` function is again long enough that a segmented presentation of its operation is warranted. The algorithm is similar to the `scan_to_bgn_match()` function earlier described (Section 7.24), except that copying replaces scanning. The function and data declarations are

```
/************************************************************************/
        BOOL
copy_to_end_match(modulestream, end_match, endcount, endoffset)
        /* Copy lines from modulestream to tgt_stream, up until the end-match
           condition is satisfied.  Return TRUE if an error was encountered,
           printing the appropriate error message; return FALSE otherwise.
/*----------------------------------------------------------------------*/
FILE    *modulestream;
STRING  end_match[ ];
int     endcount, endoffset;
{
        int     count,
                error,
                offset,
                n,
                qbx,
                qex;
        long    qb,
                qe;
        CHAR    text[MAX_LINE];
        STRING  txqueue[MAX_Q];
```

The variables **qb**, **qe**, **qbx**, **qex**, and **error** operate as they did earlier; **count** is there to measure whether any copying actually takes place. Unlike the earlier function, which maintained a file position queue, this function maintains a real text queue, `txqueue[ ]`, because excessive disk thrashing would result if a file-position queue approach function were taken, as was done in the `scan_to_bgn_match()` function described earlier (Section 7.24).

The function is otherwise very similar in structure to the scan function. The end of the queue is initialized to -1 because **qe** is incremented before it is used; therefore the first element read will be inserted into queue slot 0. A loop gets (and counts), queues, and copies text strings from the **modulestream**. When an end matching context has been specified on entry, `end_match[n]` will be **NIL** at the exit from the loop if the condition has been fulfilled. Hence, a non-**NIL** match string signals an error condition. Similarly, a zero **count** means nothing was copied—another error condition.

```
        qe = -1;
        n = 0;
        for (count = error = offset = 0; fgetstr(text, modulestream); count++)
        {       if (endoffset <= 0)

                ...
        }
        if (end_match[0] AND end_match[n])
        {       error_message(END_MATCH_ERR, end_match[n], TRUE);
                error = TRUE;
        }
        if (NOT count)
        {       error_message(NO_COPY_ERR, "", TRUE);
                error = TRUE;
        }
        return error;
}
```

There are several if-clauses inside the loop above. The first is exercised when there is a negative endoffset

```
        {       if (endoffset <= 0)
                {       qex = (int)(++qe MOD MAX_Q);
                        if (NOT (txqueue[qex] = strdup(text)))
                        {       error_message(MEMORY_ERR, "", FALSE);
                                error = TRUE;
                                break;
                        }
                        if ((qb = qe + endoffset) >=0)
                        {       qbx = (int)(qb MOD MAX_Q);
                                free(putline(txqueue[qbx]));
                                txqueue[qbx] = NIL;
                        }
                        if (end_match[n])
                        {       if (NOT strstr(text, end_match[n]))
                                        continue;

                                if (end_match[++n])
                                        continue;

                                if (--endcount <= 0)
                                {       while (qb <= qe)
                                        {       qbx = (int)(qb++ MOD MAX_Q);
                                                free(txqueue[qbx]);
                                        }
                                        break;
                                }
                                else
                                        n = 0;
                        }
                }
```

A zero or negative **endoffset** causes the text to be inserted into the next queue slot (if there is no memory error). If the queue has reached a length such that the line at the beginning of the queue is ready to be transferred into the target file, it is copied via **putline()** (Section 7.27), its memory allocation is freed, and its former queue element set to **NIL**.

If an **end_match[n]** context exists and appears within the **text** string, the next **end_match[++n]** context match is sought. When all individual end matches have been made, **endcount** is decremented and the next series of context matches are begun. When the count reaches zero, any enqueued text is freed, and the loop terminates.

The remaining **if**-clauses inside the loop operate when **endoffset** is positive:

```
else if (end_match[n])
{       putline(text);
        if (NOT strstr(text, end_match[n]))
                continue;

        if (end_match[++n])
                continue;

        if (--endcount > 0)
                n = 0;
}
else if (offset++ < endoffset)
        putline(text);
else
        break;
```

In the first of these clauses, if there is an unsatisfied ending string match, then the text is put to the target file and the line is checked for a string match. If there is a match, and if further matches are pending, the process continues. Otherwise, the **endcount** is decremented. If there are still more contexts to match, **n** is reset to 0, and the next context is sought. When **endcount** reaches zero, the end-matching context is fulfilled.

Once the end string match condition has been satisfied, the offset is checked; if that part of the end condition is unfulfilled, the text copies into the target file and is counted in **offset**. Finally, on reaching the goal **endoffset**, the end condition is satisfied, and the loop terminates.

7.26 The fgetstr() Function

The purpose of the **fgetstr()** function is to obtain a string from a named stream, replace **tab** characters in that string with spaces, and pass that string back to the calling function, **copy_to_end_match()** (Section 7.25). In addition, **fgetstr()** removes trailing white space and replaces the ending newline.

An entire line at a time is fetched from the stream, unless an end-of-file or anomalous condition has been reached. As each character of the input stream is scanned two checks are made: If a scanned character is a **tab** (i.e., `'\t'`), a span of spaces is inserted to align the next character on a multiple of the **tab_width**. If a scanned character is not a space or other printable character, the **garbage** character is substituted in its place. Otherwise, the character is written into the parameter string intact.

```
/************************************************************************/
        STRING
fgetstr(s, stream)              /* Get string s from the named stream with
                                   TABs replaced by spaces, and return it.
/*----------------------------------------------------------------------*/
STRING  s;
FILE *  stream;
{
        CHAR    c,
                p[MAX_LINE];
        int     i,
                j;
        STRING  q,
                t;

        if (NOT (q = fgets(p, MAX_LINE, stream)))
                return NIL;

        for (t = s, i = 0; c = *q++; )
        {       if (c IS '\t')
                {       j = tabwidth - (i MOD tabwidth);
                        while (j--)
                        {       *t++ = ' ';
                                ++i;
                        }
                        continue;
                }
                else if (NOT (isspace(c) OR isprint(c)))
                        c = garbage;
                *t++ = c;
                i++;
        }
        *t = NUL;
        return strcat(strtrim(s), "\n");
}
```

7.27 The putline() Function

The putline() function writes its string argument, column-adjusted in accord with the last %column directive, to its stream argument, provided that the string is not null and contains a newline. Null lines and lines not terminated in newline are both considered anomalous.

```
/***************************************************************************/
        STRING
putline(s)      /* Write the the string s onto the output stream, properly
                   columnated.  The string is presumed to exist and contain
                   a newline.  Count the output both as one of the
                   access_lines and tgt_lines.
/*-------------------------------------------------------------------------*/
STRING  s;
{
        STRING  t;

        if (s AND (t = strchr(s, '\n')))
        {       t = s + (column < 0 ? min(-column, t - s) : 0);
                fprintf(tgt_stream, "%s%s", spaces, t);
                access_lines++;
                tgt_lines++;
        }
        return s;
}
```

The local string variable t locates the newline, so t - s is the string length, not counting the newline. If the column level is negative, the input string is to be left-truncated by -column characters (the spaces string in this case will have length 0, set in response to the %column directive). This is done by starting the actual printing at the proper offset later in s. In order that the s not be accessed beyond the end point, the lesser of the two length alternatives is assigned to t. If column is zero or positive, spaces has been set to this length either by default or by a %column directive; t is set to s in this case.

The writing of t preceded by spaces thus achieves the desired formatting: For negative column values, spaces is null, and if column is zero or positive, spaces has this width.

The access_lines and tgt_lines values are augmented to count the line written both as one emanating from the access source as well as one written to the target file.

8 TOP-C LIBRARY FUNCTIONS

Several calls to functions in my personal library appear in the ConJoin program. Listings of these appear in this section for informational and completeness purposes only. None of these falls under the jurisdiction of the JPL copyright notice placed on the source files or the copyright notice printed in this report.

These files all access appropriate **#include** files to assure that each ANSI and TOP-C function reference conforms with its prototype declaration.

8.1 The stratrim() Function

The **stratrim()** function was patterned after the **atrim** function of Clipper: it removes all leading and trailing white space of a given string and returns the result.

```
/*************************************************************************/
        STRING
stratrim(s)         /* Trim all white space from s, leading and trailing, and
                       then return s.
/*-----------------------------------------------------------------------*/
STRING s;
{
        FAST STRING p;

        p = strfnb(s);
        strtcpy(s, p, strlen(p));
        return strtrim(s);
}
```

First, p advances to the first nonblank character of **s**; then **memmove()** copies p into **s**, which left-justifies **s**. Finally, trailing white space is removed from the end of the string. Copying uses **memmove()** because it properly handles overlapping areas of string arguments.

8.2 The strdup() Function

This function originally appeared in Kernighan and Ritchie [26] under the name **strsave()**. The same function later appeared in an ANSI C Standard draft document and in various C run time libraries under the name **strdup()**. My current Standard draft [27] has deleted this function, so I wrote my own, to make sure I could port things from one C to another.

```
/*****************************************************************************/
        STRING
strdup(s)               /* get enough storage for s and put s there,
                           return pointer to the string, or NIL if no
                           string space.
/*---------------------------------------------------------------------------*/
const STRING s;
{
        STRING p = NIL;

        if (s AND (p = malloc(strlen(s) + 1)))
                strcpy(p, s);
        return p;
}
```

8.3 The strfnb() Function

This simple function just skips any white space that appears at the beginning of the string parameter and returns a pointer to the first nonblank character (or to the NUL, if no nonblanks exist, or returns NIL if the given string does not exist). The function is similar to **stratrim()**, but the latter actually removes white space; **strfnb()** does not.

```
/****************************************************************************/
        STRING
strfnb(s)               /* Return a pointer to the first nonblank character
                           in string s, or to the NUL if s is empty.
/*------------------------------------------------------------------------*/
STRING  s;
{
        if (s)
                while (isspace(*s))
                        ++s;
        return s;
}
```

8.4 The strinsert() Function

This function inserts one string at the beginning of another. The algorithm is simplified using **memmove()**, which is guaranteed to transfer characters despite a possible memory overlap. First the contents of **s** are shifted right by the length of **t**. Moving one character more than the length of **s** ensures that the **s** will still end in **NUL**. Then, **t** is inserted at the beginning of **s**:

```
/*************************************************************************/
        STRING
strinsert(s, t)         /* Insert string t in string s at the beginning.
                           Return a pointer to s.
/*-----------------------------------------------------------------------*/
STRING  s, t;
{
        size_t  n;

        memmove(s + (n = strlen(t)), s, strlen(s) + 1);
        memcpy(s, t, n);
        return s;
}
```

It is noteworthy here that the **memcpy()** function copies only the **n** characters of **t**, and not its terminating **NUL**, into **s**.

8.5 The strlwr() Function

This function rewrites the given string using lowercase alphabetic characters wherever capitals appear. Some C libraries contain this non-ANSI function, while others do not. To promote portability, I wrote my own version.

```
/*************************************************************************/
          STRING
strlwr(s)        /* Set s to lowercase and return s.

/*---------------------------------------------------------------------*/
STRING s;
{
          STRING p;

          if (s)
                    for (p = s; *p = tolower(*p); p++)
                              ;
          return s;
}
```

8.6 The strnset() Function

This function replicates a given character a specified number of times into a string parameter, and returns a pointer to this string. This function also appears in some C libraries, but not in all, so I included it in mine, just to make sure.

```
/****************************************************************************/
        STRING
strnset(s, c, n)        /* Return string s composed of n character c's.

/*------------------------------------------------------------------------*/
STRING s;
size_t n;
{
        STRING p;

        for (p = s; n-- > 0; *p++ = (CHAR) c)
                ;
        *p = NUL;
        return s;
}
```

8.7 The strtcpy() Function

I wrote **strtcpy()** to correct a misconception I had about what the ANSI **strncpy()** function does. I believed that it copied one string into another for a maximum of **n** characters, returning the copied string. It does not. Rather, **strncpy(s, t, n)** copies **n** characters from string **t** to string **s** and then stops. If no terminating **NUL** is encountered, none is copied. Thus, if **n** is greater than the length of the old **s**, but shorter than the length of **t**, then the **NUL** in **s** is overwritten and none is copied from **t**. The result is that **s** has been turned into garbage. My **strtcpy()**, on the other hand, is a truncated copy that always puts in a terminating **NUL** into **s**.

```
/************************************************************************/
        STRING
strtcpy(s, t, n)        /* Truncated string copy.  Copy at most n CHARs
                           of t into s, and return s.  Note:  in contrast
                           to strncpy(), the returned copied s always ends
                           in NUL.
/*----------------------------------------------------------------------*/
STRING          s;
const STRING    t;
size_t          n;
{
        size_t m;

        if (s)
        {       m = strlen(t);
                n = MIN(n, m);
                memmove(s, t, n);
                *(s + n) = NUL;
        }
        return s;
}
```

C-2

8.8 The strtrim() Function

The **strtrim()** function is invoked directly by **fgetstr()** of Section 7.26, and also by **stratrim()** of Section 8.1. No algorithmic explanation is deemed necessary.

```
/****************************************************************************/
        STRING
strtrim(s)      /* Trim trailing white space from s and return s.

/*------------------------------------------------------------------------*/
STRING s;
{
        FAST STRING p;

        if (s AND *s)
                for (p = strlch(s); isspace(*p) AND p >= s; p--)
                        *p = NUL;
        return s;
}
```

9 MAINTAINING THE ConJoin PROGRAM

The ConJoin system described here—documents and code—are automatically built by **.MAK** files (See Appendix B). When it is time to process a report or other document within the system, the user engages the **MAKE** facility, which in turn calls the compiler, TEX, and other special processors that create updated products.

These files may require alteration to conform with the user's development environment, whenever a different compiler, directory structure, or set of utilities is present.

10 ERROR MESSAGES

Error messages take the form

file line offset message

where *file* names the source file in which the error occurs, *line* is the offending line, *offset* is the approximate character on *line* where the violation occurs, and *message* is a diagnostic message. Such error forms can be used by some text editors for automatically placing the cursor on the lines for correction.

(1) **`Beginning match string not found:`** *match_string.*

The given *match_string* could not be found in the file named on the `%access` directive.

(2) **`Break string is invalid:`** *break.*

An invalid *break* string has been detected. This occurs when no instance of the *break* string is found on the **`%access`** directive line.

(3) **`Command line error:`**

An error on the invoking command line has been detected. File name and location information is omitted. Causes of this message are

- **`Unknown option.`** The command line has a string argument beginning in `'/'` or `'-'` that is not recognized as an option.
- **`Unknown command.`** The command line has a string argument unrecognized as either source or target file name.
- **`No such file or directory.`** The file cannot be located.

(4) **`End-match string not found:`** *match_string.*

The given ending *match_string* could not be found in the file named on the `%access` directive.

(5) **`Input and output files may not be the same:`** *name.*

The target file is not permitted to overwrite the source file. File name and location information is omitted from the message.

(6) **`Memory insufficient for queue`**.

Memory is unavailable from **`malloc()`** to queue the **`%access`** stream. **`MAX_Q`** may be set too large, or too many "terminate and stay resident" programs may be in memory. Try removing some of the TSR's, or failing that, recompiling *ConJ*`oin` with a smaller **`MAX_Q`**.

(7) **`No access file found:`** *name.*

The file *name* given in the **`%access`** directive could not be found. Check the file name, its spelling, and **`%path`** directives for locatability.

(8) **`No lines copied from accessed file.`**

`%access` has failed to transcribe anything from the file named in the directive. This often happens as a result of misspelling in the starting *match_string*. The error message is also put into the target file.

(9) **`Range separator missing.`**

The `%access` directive has the *range_separator* absent between the two match-strings.

(10) **`Size command case invalid:`** *case.*

A `%size` case other than `C`, `a`, `T` or `r` has been detected. A zero is substituted into the text at this point.

APPENDICES

A *ConJoin* PROGRAM LISTING

```
#define VERSION "(08-Apr-1992)" /*                              (ConJoin.c)*/
char copyrightnotice[14][76] =
{  "##########################################################################",
   "#                                                                        #",
   "#         Copyright (C) 1992, California Institute of Technology         #",
   "#      All rights reserved.  U. S. Government sponsorship under NASA     #",
   "#                    Contract NAS7-918 is acknowledged.                  #",
   "#                                                                        #",
   "#                        Robert C. Tausworthe                            #",
   "#                      Jet Propulsion Laboratory                         #",
   "#                        4800 Oak Grove Drive                            #",
   "#                      Pasadena, CA 91109-8099                           #",
   "#                                                                        #",
   "#                                                                        #",
   "##########################################################################",
  ""
};
/*
                        ANSI STANDARD HEADER FILES                      */

#include <ctype.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>


/*-----------------------------------------------------------------------*

                        DEFINITIONS                                     */

typedef int              BOOL;
typedef unsigned char   CHAR;
typedef unsigned char *  STRING;

#define GLOBAL           extern
#define LOCAL            static

#define NIL              ((void *) 0)
#define NUL              0

#define FALSE            0
#define TRUE             1
```

```
#define AND             &&
#define IS              ==
#define NOT             !
#define MOD             %
#define OR              ||

#ifndef FILENAME_MAX
#define FILENAME_MAX     50
#endif
#define MAX_CONTEXT      10
#define MAX_LINE        135
#define MAX_Q           100
#define NUMBER_OF_PATHS  20
#define PAGE_WIDTH       75

#define OPEN_COMMENT    "%"
#define CLOSE_COMMENT   ""
#define COMMENT_LENGTH  10
#define ACCESS_SIGNAL   OPEN_COMMENT "access"
#define BREAK_SIGNAL    OPEN_COMMENT "break"
#define COLUMN_SIGNAL   OPEN_COMMENT "column"
#define COUNT_SIGNAL    OPEN_COMMENT "count"
#define GARBAGE_SIGNAL  OPEN_COMMENT "garbage"
#define PATH_SIGNAL     OPEN_COMMENT "path"
#define POSTFIX_SIGNAL  OPEN_COMMENT "postfix"
#define PREFIX_SIGNAL   OPEN_COMMENT "prefix"
#define RANGE_SIGNAL    OPEN_COMMENT "range"
#define SHOW_SIGNAL     OPEN_COMMENT "show"
#define SIZE_SIGNAL     OPEN_COMMENT "size"
#define TAB_SIGNAL      OPEN_COMMENT "tabs"

#define BREAK_DEFAULT    ","
#define COLUMN_DEFAULT   0
#define COUNT_DEFAULT    "#"
#define GARBAGE_DEFAULT  '#'
#define OLD_TYPE_DEFAULT ".T_X"
#define POSTFIX_DEFAULT  "\\end" "{verbatim}}"
#define POSTVERBATIM     "|\n\n"
#define PREFIX_DEFAULT   "{\\footnotesize \\begin{verbatim}"
#define PREVERBATIM      "\\noindent \\verb|"
#define RANGE_DEFAULT    "-"
#define SHOW_DEFAULT     FALSE
#define TAB_DEFAULT      8
#define TGT_TYPE_DEFAULT ".TeX"

#define BGN_MATCH_ERR   0
#define BREAK_ERR       1
#define CMD_LINE_ERR    2
#define END_MATCH_ERR   3
#define IO_SAME_ERR     4
#define MEMORY_ERR      5
#define NO_ACCESS_ERR   6
#define NO_COPY_ERR     7
#define RANGE_ERR       8
#define SIZE_ERR        9
```

```
#define FGETSIZE(s)      strdup(strtrim(fgets(s, MAX_LINE, size_stream)))

/*--------------------------------------------------------------------*

                         TOP-C FUNCTION PROTOTYPES                   */

GLOBAL STRING            stratrim(STRING);
GLOBAL STRING            strdup(STRING);
GLOBAL STRING            strfnb(STRING);
GLOBAL STRING            strinsert(STRING, STRING);
GLOBAL STRING            strlwr(STRING);
GLOBAL STRING            strnset(STRING, int, int);
GLOBAL STRING            strtcpy(STRING, STRING, int);
GLOBAL STRING            strtrim(STRING);


/*--------------------------------------------------------------------*

                         FUNCTION PROTOTYPES                         */

int                      main(int, STRING [ ]);

LOCAL BOOL               access(STRING);
LOCAL BOOL               access_condition(STRING, STRING [ ], int *, int *, int);
LOCAL void               announce(void);
LOCAL void               command_line(int, STRING [ ], STRING);
LOCAL BOOL               ConJoin_files(void);
LOCAL BOOL               copy_to_end_match(FILE *, STRING [ ], int, int);
LOCAL BOOL               directive(STRING, STRING);
LOCAL void               error_message(int, STRING, BOOL);
LOCAL STRING             fgetstr(STRING, FILE *);
LOCAL void               file_defaults(void);
LOCAL FILE *             file_open(STRING, STRING);
LOCAL STRING             right_fill(STRING, int);
LOCAL BOOL               match_parameters(STRING, STRING, STRING [ ], STRING [ ],
                            int *, int *, int*, int *);
LOCAL FILE *             open_access(STRING);
LOCAL void               open_io_files(void);
LOCAL void               initialization(int, STRING [ ]);
LOCAL STRING             putline(STRING);
LOCAL BOOL               scan_to_bgn_match(FILE *, STRING [ ], int, int);
LOCAL STRING             strext(STRING, STRING);
LOCAL void               timestamp(void);
LOCAL void               usage(void);


/*--------------------------------------------------------------------*

                         LOCAL DATA STRUCTURES                       */

LOCAL long    access_lines;
LOCAL CHAR    close[COMMENT_LENGTH]       = CLOSE_COMMENT;
LOCAL int     column                      = COLUMN_DEFAULT;
LOCAL CHAR    ConJoin_file[FILENAME_MAX]  = "";
LOCAL long    ConJoin_lines;
LOCAL FILE *  ConJoin_stream              = NIL;
LOCAL STRING  countsignal                 = COUNT_DEFAULT;
LOCAL BOOL    credits                     = TRUE;
LOCAL CHAR    garbage                     = GARBAGE_DEFAULT;
LOCAL STRING  last_acc_lines              = NIL;
LOCAL STRING  last_CJn_lines              = NIL;
LOCAL STRING  last_tgt_lines              = NIL;
LOCAL STRING  last_use_lines              = NIL;
LOCAL CHAR    mark[10]                    = BREAK_DEFAULT;
LOCAL CHAR    open[COMMENT_LENGTH]        = OPEN_COMMENT;
LOCAL STRING  path_list[NUMBER_OF_PATHS];
LOCAL int     path_list_size              = 0;
LOCAL CHAR    postfix[MAX_LINE]           = POSTFIX_DEFAULT;
LOCAL CHAR    postverbatim[30]            = POSTVERBATIM;
LOCAL CHAR    prefix[MAX_LINE]            = PREFIX_DEFAULT;
LOCAL CHAR    preverbatim[30]             = PREVERBATIM;
LOCAL CHAR    range[10]                   = RANGE_DEFAULT;
LOCAL BOOL    show                        = SHOW_DEFAULT;
LOCAL CHAR    size_file[FILENAME_MAX]     = "";
LOCAL CHAR    spaces[MAX_LINE]            = "";
LOCAL int     tabwidth                    = TAB_DEFAULT;
LOCAL CHAR    tgt_file[FILENAME_MAX]      = "";
LOCAL long    tgt_lines;
LOCAL FILE *  tgt_stream                  = NIL;
                                                                /* main */
/***************************************************************************/

main(argc, argv)        /* Process a ConJoin file to create a target file.
                           Return a FALSE value if no failure occurs, or
                           TRUE or other nonzero value if a failure was
                           encountered.
/*-----------------------------------------------------------------------*/
STRING argv[ ];
{
        BOOL    failure;
        FILE    *size_stream;
        int     i;

        initialization(argc, argv); /* terminates if no source file named     */
        open_io_files();            /* terminates on failure in opening files */
        access_lines = ConJoin_lines = tgt_lines = 0;
        timestamp();
        failure = ConJoin_files();
        failure |= fclose(ConJoin_stream) | fclose(tgt_stream);
        for (i = 0; i < path_list_size; i++)
                free(path_list[i]);
        free(last_acc_lines);
        free(last_CJn_lines);
        free(last_tgt_lines);
        free(last_use_lines);
        printf("Processed:\n%10ld %s source lines\n%10ld accessed lines\n"
            "%10ld %s total lines written\n", ConJoin_lines, ConJoin_file,
            access_lines, tgt_lines, tgt_file);
        if (size_stream = fopen(size_file, "w"))
        {       fprintf(size_stream, "%ld\n%ld\n%ld\n%.3f\n", ConJoin_lines,
                    access_lines, tgt_lines,
                    (double) access_lines / (double) tgt_lines);
                failure |= fclose(size_stream);
        }
        return failure;
}
                                /* end main */
                                                        /* initialization */
/*************************************************************************/
        void
initialization(argc, argv)
        /* Process command line file names and options, and retrieve
           size_file statistics.
/*-----------------------------------------------------------------------*/
STRING argv[ ];
{
        CHAR    msg[MAX_LINE];
        STRING  s;
        FILE    *size_stream;

        command_line(argc, argv, msg);
        announce();
        if (*msg)
        {       usage();
                error_message(CMD_LINE_ERR, msg, FALSE);
                exit(TRUE);
        }
        file_defaults();
        if (NOT strcmp(ConJoin_file, tgt_file))
        {       strcpy(msg, ConJoin_file);
                *ConJoin_file = NUL;
                usage();
                error_message(IO_SAME_ERR, msg, FALSE);
                exit(TRUE);
        }
        strcpy(strchr(s = strcpy(size_file, tgt_file), '.'), ".siz");
        if (size_stream = fopen(size_file, "r"))
        {       last_CJn_lines = FGETSIZE(msg);
                last_acc_lines = FGETSIZE(msg);
                last_tgt_lines = FGETSIZE(msg);
                last_use_lines = FGETSIZE(msg);
                fclose(size_stream);
        }
}
                                /* end initialization */
                                                        /* command_line */
/*************************************************************************/
        void
command_line(argc, argv, msg)
        /* Process information on the command line: extract ConJoin_file
           and tgt_file names, and option -a, when present.  Return with
           msg set to error conditions.
/*-----------------------------------------------------------------------*/
STRING argv[ ], msg;
```

```
{
        int     i;
        STRING  s;

        *msg = NUL;
        for (i = 1; i < argc; i++)
        {       s = strlwr(argv[i]);
                if (*s IS '-' OR *s IS '/')
                {       switch (*++s)
                        {   case 'a':
                                credits = FALSE;
                                break;
                            default:
                                sprintf(msg + strlen(msg), "Unknown option: "
                                    "%s.\n", argv[i]);
                        }
                        *argv[i] = NUL;
                }
                else if (NOT *ConJoin_file)
                        strcpy(ConJoin_file, s);
                else if (NOT *tgt_file)
                        strcpy(tgt_file, s);
                else
                        sprintf(msg + strlen(msg), "Unknown command: %s.\n", s);
        }
        if (NOT *ConJoin_file)
                strcat(msg, "No source file named.\n");
}
                                /* end command_line */
                                                        /* announce */
/*************************************************************************/
        void
announce()              /* Announce program, copyright, and author.

/*-----------------------------------------------------------------------*/
{
        int     i;

        if (credits)
        {       for (i = 0; *copyrightnotice[i]; i++)
                        printf("%s\n", copyrightnotice[i]);
                printf("\n\t\t\t   ConJoin Program"
                       "\n\t\t\t    %s\n\n", VERSION);
        }
}
                                /* end announce */
                                                    /* error_message */
/*************************************************************************/
        void
error_message(n, s, f)  /* Write error message n augmented with string s
                           to stdout, indicating the current line in the
                           source file.  Repeat the message on the target
                           file if f is TRUE.
/*-----------------------------------------------------------------------*/
STRING s;
```

```
{
        LOCAL STRING  errmsg[ ] =
        {       "Beginning match string not found: ",
                "Break string is invalid: ",
                "Command line error: ",
                "End-match string not found: ",
                "Input and output files may not be the same: ",
                "Memory insufficient for queue.",
                "No access file found: ",
                "No lines copied from accessed file.",
                "Range separator missing.",
                "Size command case invalid: "
        };

        if (*ConJoin_file)
                printf("\a%s, %ld 1: %s%s\n",
                    ConJoin_file, ConJoin_lines, errmsg[n], s);
        else
                printf("%s%s\n", errmsg[n], s);
        if (f)
                fprintf(tgt_stream, "***ERROR*** %s%s\n", errmsg[n], s);
}
                                /* end error_message */
                                                          /* usage */
/************************************************************************/
        void
usage()                 /* Print a message on usage syntax of ConJoin.

/*----------------------------------------------------------------------*/
{
        printf("Usage:  ConJoin <ConJoin source> <target file> "
            "[<options>]\n\n"
            "\tSource file type default is .CJn\n"
            "\tTarget file type default is %s\n\n"
            "Options:\n"
            "\t-a      Do not announce the program.\n", TGT_TYPE_DEFAULT);
}
                                /* end usage */
                                                   /* file_defaults */
/************************************************************************/
        void
file_defaults()         /* Supply ConJoin file type .CJn if missing, and
                           supply missing parts of tgt_file, if any.
/*----------------------------------------------------------------------*/
{
        STRING  s;

        if (NOT (s = strchr(ConJoin_file, '.')))
                strcat(ConJoin_file, ".CJn");
        if (NOT *tgt_file)
        {       strtcpy(tgt_file, ConJoin_file,
                    strchr(ConJoin_file, '.') - ConJoin_file);
        }
        if (NOT (s = strchr(tgt_file, '.')))
                strcat(tgt_file, TGT_TYPE_DEFAULT);
```

```
}
                                   /* end file_defaults */
                                                    /* open_io_files */
/**********************************************************************/
        void
open_io_files()         /* Open ConJoin_file and tgt_file into ConJoin_stream
                           and tgt_stream.  Rename old tgt_file, if any,
                           with OLD_TYPE_DEFAULT.  Terminate with an error
                           message via file_open() if files cannot be opened.
/*--------------------------------------------------------------------*/
{
        CHAR    tgt_bak[FILENAME_MAX];
        FILE *  f;
        STRING  s;

        ConJoin_stream = file_open(ConJoin_file, "r");
        if (f = fopen(tgt_file, "r"))
        {       fclose(f);
                s = strchr(strcpy(tgt_bak, tgt_file), '.');
                strcpy(s, OLD_TYPE_DEFAULT);
                remove(tgt_bak);
                rename(tgt_file, tgt_bak);
        }
        tgt_stream = file_open(tgt_file, "w");
}
                                   /* end open_io_files */
                                                    /* file_open */
/**********************************************************************/
        FILE *
file_open(name, use)
        /* Open the name file for given use, and return resulting stream.
           If file cannot be opened, print reason, and abort processing.
/*--------------------------------------------------------------------*/
STRING name, use;
{
        CHAR    s[MAX_LINE];
        FILE *  stream;

        if (NOT (stream = fopen(name, use)))
        {       strcpy(s, name);
                *name = NUL;
                strncat(strcat(s, ": "), strerror(errno), MAX_LINE - 1);
                error_message(CMD_LINE_ERR, s, FALSE);
                exit(TRUE);
        }
        return stream;
}
                                   /* end file_open */
                                                    /* timestamp */
/**********************************************************************/
        void
timestamp()     /* Write the tgt_file name and a time-stamped header with
                   a revision warning onto the tgt_stream.
/*--------------------------------------------------------------------*/
{
        time_t          clock;
        CHAR            atime[26],
                        bar[MAX_LINE],
                        blanks[MAX_LINE];
        STRING          sp, text;
        struct tm *     t;
        int             n;

        n = strlen(open) + strlen(close);
        strnset(bar, '=', PAGE_WIDTH - n);
        strnset(blanks, ' ', PAGE_WIDTH - n - 2);
        time(&clock);
        t = localtime(&clock);
        stratrim(strcpy(atime, asctime(t)));
        sp = right_fill(atime, strlen(tgt_file) + n + 5);
        fprintf(tgt_stream, "%s (%s)%s(%s)%s\n", open, atime, sp,
            tgt_file, close);
        fprintf(tgt_stream, "%s%s%s\n", open, bar, close);
        text = "%s|        This file was ConJoin-ed from input file %s.%s|%s\n";
        sp = right_fill(text, strlen(ConJoin_file) - n - 7);
        fprintf(tgt_stream, text, open, ConJoin_file, sp, close);
        fprintf(tgt_stream, "%s|%s|%s\n", open, blanks, close);
        text = "%s|                DO NOT REVISE THIS FILE.%s|%s\n";
        fprintf(tgt_stream, text, open, right_fill(text, n - 7), close);
        fprintf(tgt_stream, "%s|%s|%s\n", open, blanks, close);
        text = "%s|        To make revisions, modify the original file.%s|%s\n";
        fprintf(tgt_stream, text, open, right_fill(text, n - 7), close);
        fprintf(tgt_stream, "%s%s%s\n", open, bar, close);
        tgt_lines += 8;
}
                                /* end timestamp */

                                                        /* right_fill */
/*************************************************************************/
        STRING
right_fill(s, n)        /* Generate spaces as a blank string of length
                           PAGE_WIDTH - n.  Return spaces.
/*-----------------------------------------------------------------------*/
STRING  s;
{
        return strnset(spaces, ' ', PAGE_WIDTH - strlen(s) - n);
}
                                /* end right_fill */
                                                     /* ConJoin_files */
/*************************************************************************/

ConJoin_files()         /* Process the ConJoin source file and create the
                           expanded target file.  Return a nonzero value
                           if an error occurs, or a 0 value if none.  Count
                           ConJoin_lines, access_lines, and tgt_lines.
/*-----------------------------------------------------------------------*/
{
        BOOL    error;
        CHAR    hold[MAX_LINE],
                line[MAX_LINE];
        STRING  extract;

        error = FALSE;
        *spaces = NUL;
        for (ConJoin_lines = 0; fgets(line, MAX_LINE, ConJoin_stream); )
        {       ConJoin_lines++;
                tgt_lines++;
                if (NOT *strfnb(line))
                {       fprintf(tgt_stream, "%s", line);
                        continue;
                }
                if (strstr(line, open))
                {       if (extract = strstr(line, ACCESS_SIGNAL))
                                strcpy(hold, line);
                        error |= directive(extract, line);
                }
                else
                        extract = NIL;
                fprintf(tgt_stream, "%s", line);
                if (extract)
                        error |= access(hold + (extract - line) +
                            strlen(ACCESS_SIGNAL));
        }
        return error;
}
                                /* end ConJoin_files */
                                                        /* directive */
/*************************************************************************/
        BOOL
directive(extract, text)
        /* Process ConJoin directives that may appear in the text string.
           If extract is non-NIL, the text contains an access directive
           that may only need to be prepared for show-ing.  In case of
           %size, write the appropriate values into text.  Insert
           preverbatim and postverbatim into text if show is, or has just
           turned, TRUE.  Return TRUE if a bad %size case appears; FALSE
           otherwise.
/*-----------------------------------------------------------------------*/
STRING extract, text;
{
        STRING  s,
                t;
        CHAR    g,
                line[MAX_LINE];
        BOOL    change_show;

        strcpy(line, text);
        if (extract)
                t = line + (extract - text);
        else if (t = strstr(line, BREAK_SIGNAL))
                strcat(mark, t + strlen(BREAK_SIGNAL));
        else if (t = strstr(line, COLUMN_SIGNAL))
        {       column = atoi(t + strlen(COLUMN_SIGNAL));
                if (column > 0)
                {       strnset(spaces, ' ', column);
                column = 0;
        }
        else
                *spaces = NUL;
}
else if (t = strstr(line, COUNT_SIGNAL))
        strext(countsignal, t + strlen(COUNT_SIGNAL));
else if (t = strstr(line, GARBAGE_SIGNAL))
{       if (g = *strfnb(t + strlen(GARBAGE_SIGNAL)))
                garbage = g;
}
else if (t = strstr(line, PATH_SIGNAL))
{       path_list[path_list_size++] =
            strdup(strext(t, t + strlen(PATH_SIGNAL)));
}
else if (t = strstr(line, POSTFIX_SIGNAL))
        strext(postfix, t + strlen(POSTFIX_SIGNAL));
else if (t = strstr(line, PREFIX_SIGNAL))
        strext(prefix, t + strlen(PREFIX_SIGNAL));
else if (t = strstr(line, RANGE_SIGNAL))
        strext(range, t + strlen(RANGE_SIGNAL));
else if (t = strstr(line, SHOW_SIGNAL))
{       s = strlwr(stratrim(t + strlen(SHOW_SIGNAL)));
        if (NOT strcmp(s, "on"))
                change_show = TRUE + TRUE;
        else if (NOT strcmp(s, "off"))
                change_show = TRUE + FALSE;
        if (change_show AND NOT show)
                show = change_show - TRUE;
}
else if (t = strstr(line, SIZE_SIGNAL))
{       do
        {       *t = NUL;
                switch (*(t = strfnb(t + strlen(SIZE_SIGNAL))))
                {   case 'c':
                    case 'C':
                        s = last_CJn_lines;
                        break;
                    case 'a':
                    case 'A':
                        s = last_acc_lines;
                        break;
                    case 'r':
                    case 'R':
                        s = last_use_lines;
                        break;
                    case 't':
                    case 'T':
                        s = last_tgt_lines;
                        break;
                    default:
                        s = "0";
                        error_message(SIZE_ERR, text, FALSE);
                        return TRUE;
                }
```

```
                        sprintf(text, "%s%s%s", line, s, ++t);
                } while (t = strstr(strcpy(line, text), SIZE_SIGNAL));
        }
        else if (t = strstr(line, TAB_SIGNAL))
                tabwidth = atoi(t + strlen(TAB_SIGNAL));
        if (t AND show)
        {       strinsert(text + (t - line), preverbatim);
                if (*close AND (t = strstr(text, close)))
                        strinsert(text, postverbatim);
                else
                        strcat(stratrim(text), postverbatim);
                if (change_show)
                        show = change_show - TRUE;
        }
        return FALSE;
}
                                /* end directive */
                                                          /* strext */
/*************************************************************************/
        STRING
strext(s, t)    /* Extract string t up to the close-comment string, if close
                   is not null, or to the end of t if null, into s. Remove
                   leading and trailing blanks from s and return s.
/*-----------------------------------------------------------------------*/
STRING  s, t;
{
        STRING p;

        if (*close AND (p = strstr(t, close)))
                strtcpy(s, t, p - t);
        else
                strcpy(s, t);
        return stratrim(s);
}
                                /* end strext */
                                                          /* access */
/*************************************************************************/
        BOOL
access(buffer)          /* Process the text extraction operation specified
                           in the line buffer to the tgt_stream.  Return
                           FALSE if no error, TRUE if an error occurred.
/*-----------------------------------------------------------------------*/
STRING  buffer;
{
        STRING  bgn_match[MAX_CONTEXT + 1],
                end_match[MAX_CONTEXT + 1];
        CHAR    module[FILENAME_MAX];
        BOOL    error;
        FILE *  modulestream;
        int     bgncount,
                bgnoffset,
                endcount,
                endoffset,
                i;
        strext(buffer, buffer);
        if (match_parameters(buffer, module, bgn_match, end_match, &bgncount,
            &bgnoffset, &endcount, &endoffset))
                return TRUE;

        if (NOT (modulestream = open_access(module)))
        {       error_message(NO_ACCESS_ERR, module, FALSE);
                return TRUE;
        }
        printf("%saccess %s%s\n", open, buffer, close);
        if (*prefix)
                fprintf(tgt_stream, "%s\n", prefix);
        error  = scan_to_bgn_match(modulestream, bgn_match, bgncount,
            bgnoffset);
        error |= copy_to_end_match(modulestream, end_match, endcount,
            endoffset);
        for (i = 0; bgn_match[i]; i++)
                free(bgn_match[i]);
        for (i = 0; end_match[i]; i++)
                free(end_match[i]);
        error |= fclose(modulestream);
        if (*postfix)
                fprintf(tgt_stream, "%s\n", postfix);
        return error;
}
                                /* end access */
                                                        /* match_parameters */
/**********************************************************************/
        BOOL
match_parameters(buffer, module, bgn_match, end_match, bgncount, bgnoffset,
    endcount, endoffset)
        /* Extract access module name, and beginning and ending access
           conditions.  Return TRUE if an error is encountered, FALSE
           otherwise.
/*--------------------------------------------------------------------*/
STRING  buffer, bgn_match[ ], end_match[ ], module;
int     *bgncount, *bgnoffset, *endcount, *endoffset;
{
        CHAR    line[MAX_LINE];
        STRING  s,
                t;

        if (NOT (s = strstr(strcpy(line, buffer), mark)))
        {       error_message(BREAK_ERR, buffer, FALSE);
                return TRUE;
        }
        stratrim(strtcpy(module, line, s - line));
        if (NOT (*(t = strfnb(s + strlen(mark)))
            AND (s = strstr(t, range))))
        {       error_message(RANGE_ERR, buffer, FALSE);
                return TRUE;
        }
        *s = NUL;
        s += strlen(range);
        if (access_condition(t, bgn_match, bgncount, bgnoffset, 1))
```

```
                return TRUE;

        if (access_condition(s, end_match, endcount, endoffset, -1))
                return TRUE;

        return FALSE;
}
                        /* end match_parameters */
                                                        /* access_condition */
/**************************************************************************/
        BOOL
access_condition(buffer, match, count, offset, init)
        /* Extract access condition from buffer. Set offset to init if
           no offset is parsed in the buffer.  Return TRUE if an error
           is encountered, FALSE otherwise.
/*------------------------------------------------------------------------*/
STRING  buffer, match[ ];
int     *count, *offset;
{
        int     n;
        STRING  s, t;

        *offset = init;
        *count = 1;
        s = stratrim(buffer);
        if (t = strstr(s, countsignal))
        {       *count = atoi(s = t + strlen(countsignal));
                *t = NUL;
        }
        if ((t = strchr(s, '+')) OR (t = strchr(s, '-')))
        {       *offset = atoi(t);
                *t = NUL;
        }
        else if (isdigit(*buffer))
        {       *offset = atoi(buffer);
                *buffer = NUL;
        }
        s = buffer;
        for (n = 0; *s AND n < MAX_CONTEXT; n++)
        {       if (t = strstr(s, mark))
                {       *t = NUL;
                        t += strlen(mark);
                }
                match[n] = strdup(strtrim(s));
                s = strfnb(t AND *t ? t : s + strlen(s));
        }
        if (n > 0 AND match[n - 1] IS NIL)
                return TRUE;

        match[n] = NIL;
        return FALSE;
}
                        /* end access_condition */
                                                        /* open_access */
/**************************************************************************/
```

```
        FILE *
open_access(file)       /* Open the specified file for reading.

/*-----------------------------------------------------------------------*/
STRING  file;
{
        int     i;
        CHAR    path[FILENAME_MAX];
        FILE *  stream;

        strcpy(path, file);
        for (i = 0; NOT (stream = fopen(path, "r")) AND i < path_list_size; i++)
                strcat(strcpy(path, path_list[i]), file);
        return stream;
}
                                /* end open_access */
                                                /* scan_to_bgn_match */
/*************************************************************************/
        BOOL
scan_to_bgn_match(modulestream, bgn_match, bgncount, bgnoffset)
        /* Scan the modulestream for the beginning match condition and
           return positioned to access the first line to be copied.
/*-----------------------------------------------------------------------*/
FILE *  modulestream;
STRING  bgn_match[ ];
int     bgncount, bgnoffset;
{
        fpos_t  fp,
                fpqueue[MAX_Q];
        int     n,
                offset,
                qex;
        BOOL    error;
        long    qe;
        CHAR    text[MAX_LINE];

        n = error = offset = 0;
        qe      = -1;
        while (NOT fgetpos(modulestream, &fp) AND
            fgets(text, MAX_LINE, modulestream))
        {       qex = (int)(++qe MOD MAX_Q);
                fpqueue[qex] = fp;
                if (bgn_match[n])
                {       if (NOT strstr(text, bgn_match[n]))
                                        continue;

                        if (bgn_match[++n])
                                continue;

                        if (--bgncount <= 0)
                        {       if (bgnoffset <= 0)
                                {       qex = (int)((qe + bgnoffset) MOD MAX_Q);
                                        fsetpos(modulestream, &fpqueue[qex]);
                                        break;
                                }
                                else if (bgnoffset-- IS 1)
                                        break;
                        }
                        else
                                n = 0;
                }
                else if (NOT bgnoffset)
                {       fsetpos(modulestream, &fpqueue[qex]);
                        break;
                }
                else if (++offset >= bgnoffset)
                        break;
        }
        if (bgn_match[0] AND bgn_match[n])
        {       error_message(BGN_MATCH_ERR, bgn_match[n], TRUE);
                error = TRUE;
        }
        return error;
}
                        /* end scan_to_bgn_match */
                                                /* copy_to_end_match */
/************************************************************************/
        BOOL
copy_to_end_match(modulestream, end_match, endcount, endoffset)
        /* Copy lines from modulestream to tgt_stream, up until the end-match
           condition is satisfied.  Return TRUE if an error was encountered,
           printing the appropriate error message; return FALSE otherwise.
/*----------------------------------------------------------------------*/
FILE    *modulestream;
STRING  end_match[ ];
int     endcount, endoffset;
{
        int     count,
                error,
                offset,
                n,
                qbx,
                qex;
        long    qb,
                qe;
        CHAR    text[MAX_LINE];
        STRING  txqueue[MAX_Q];

        qe = -1;
        n = 0;
        for (count = error = offset = 0; fgetstr(text, modulestream); count++)
        {       if (endoffset <= 0)
                {       qex = (int)(++qe MOD MAX_Q);
                        if (NOT (txqueue[qex] = strdup(text)))
                        {       error_message(MEMORY_ERR, "", FALSE);
                                error = TRUE;
                                break;
                        }
                        if ((qb = qe + endoffset) >=0)
                        {       qbx = (int)(qb MOD MAX_Q);
```

```
                                free(putline(txqueue[qbx]));
                                txqueue[qbx] = NIL;
                        }
                        if (end_match[n])
                        {       if (NOT strstr(text, end_match[n]))
                                        continue;

                                if (end_match[++n])
                                        continue;

                                if (--endcount <= 0)
                                {       while (qb <= qe)
                                        {       qbx = (int)(qb++ MOD MAX_Q);
                                                free(txqueue[qbx]);
                                        }
                                        break;
                                }
                                else
                                        n = 0;
                        }
                }
                else if (end_match[n])
                {       putline(text);
                        if (NOT strstr(text, end_match[n]))
                                continue;

                        if (end_match[++n])
                                continue;

                        if (--endcount > 0)
                                n = 0;
                }
                else if (offset++ < endoffset)
                        putline(text);
                else
                        break;
        }
        if (end_match[0] AND end_match[n])
        {       error_message(END_MATCH_ERR, end_match[n], TRUE);
                error = TRUE;
        }
        if (NOT count)
        {       error_message(NO_COPY_ERR, "", TRUE);
                error = TRUE;
        }
        return error;
}
                        /* end copy_to_end_match */
                                                              /* fgetstr */
/**************************************************************************/
        STRING
fgetstr(s, stream)              /* Get string s from the named stream with
                                   TABs replaced by spaces, and return it.
/*----------------------------------------------------------------------*/
STRING  s;
```

```
FILE *  stream;
{
        CHAR    c,
                p[MAX_LINE];
        int     i,
                j;
        STRING  q,
                t;

        if (NOT (q = fgets(p, MAX_LINE, stream)))
                return NIL;

        for (t = s, i = 0; c = *q++; )
        {       if (c IS '\t')
                {       j = tabwidth - (i MOD tabwidth);
                        while (j--)
                        {       *t++ = ' ';
                                ++i;
                        }
                        continue;
                }
                else if (NOT (isspace(c) OR isprint(c)))
                        c = garbage;
                *t++ = c;
                i++;
        }
        *t = NUL;
        return strcat(strtrim(s), "\n");
}
                                /* end fgetstr */
                                                        /* putline */
/***********************************************************************/
        STRING
putline(s)      /* Write the the string s onto the output stream, properly
                   columnated.  The string is presumed to exist and contain
                   a newline.  Count the output both as one of the
                   access_lines and tgt_lines.
/*---------------------------------------------------------------------*/
STRING  s;
{
        STRING  t;

        if (s AND (t = strchr(s, '\n')))
        {       t = s + (column < 0 ? min(-column, t - s) : 0);
                fprintf(tgt_stream, "%s%s", spaces, t);
                access_lines++;
                tgt_lines++;
        }
        return s;
}
                                /* end putline */
/*---------------------------------------------------------------------*/
                                /* end program */
```

B *ConJo*in CONSTRUCTION

B.1 The Master Document File

```
% (30-Jul-1992)                                                    (CJ_reprt.TeX)
%#############################################################################
%#                                                                           #
%#         Copyright (C) 1992, California Institute of Technology            #
%#       All rights reserved.  U. S. Government sponsorship under NASA       #
%#                     Contract NAS7-918 is acknowledged.                    #
%#                                                                           #
%#                          Robert C. Tausworthe                             #
%#                        Jet Propulsion Laboratory                          #
%#                          4800 Oak Grove Drive                             #
%#                        Pasadena, CA 91109-8099                            #
%#                                                                           #
%#                                                                           #
%#############################################################################

%                                JPL REPORT FORMAT

\documentstyle[taus,twoside]{article} % taus.sty is needed because it
                                      % contains the box shapes for Figure 1
                                      % and the JPL Report format.

\newcommand{\path}{CJ_}

\input{\path style}

\newcommand{\Title}{Conjunctive Programming: \\ An Integrative Approach to \\
    Software System Synthesis}
\newcommand{\PubDate}{August 31, 1992}
\newcommand{\DocNumber}{92-12}
\newcommand{\work}{report}

\begin{document}

%------------------------------ COVER PAGE-------------------------------------

\pagestyle{empty}
\coverpage{JPL Publication \DocNumber}{\Title}{\Author}{\PubDate}

%------------------------------ FRONT MATTER STYLE ----------------------------

\rm
\pagestyle{myheadings}
\markboth{\DocNumber}{\DocNumber}
\pagenumbering{roman}
\setcounter{page}{2}

%------------------------------ NASA AND OTHER CREDITS ------------------------

        \input{\path credt}

%------------------------------- ABSTRACT -------------------------------------
```

```
\rectopage
\putmidpage
{       \begin{center}
                \large \bf Abstract\\[.3in] \normalsize
        \end{center}

        \begin{quotation}
                \input{\path abstr}
        \end{quotation}
}

%------------------------------ ACKNOWLEDGMENT -------------------------------

\clearpage
\putmidpage
{
        \begin{center}
                \large \bf Acknowledgement\\[.3in] \normalsize
        \end{center}

        \begin{quotation}
                \input{\path ackno}
        \end{quotation}
}

%------------------------------ TABLE OF CONTENTS ----------------------------

\rectopage
\begin{center}
        \Large \bf Contents\\[.5in]
        \normalsize
\end{center}

\Tableofcontents

%------------------------------ LIST OF FIGURES ------------------------------

\vspace*{0.5in}
\begin{center}
        {\Large \bf List of Figures}
\end{center}

\Listoffigures

%------------------------------ LIST OF TABLES -------------------------------

%       NOTE: THIS SECTION IS COMMENTED OUT BECAUSE NO TABLES CURRENTLY
%       EXIST IN THE REPORT.

%\vspace*{0.5in}
%\begin{center}
%        {\Large \bf \noindent List of Tables}
%\end{center}
```

```
%\Listoftables

%------------------------------ BODY ------------------------------------------

\rectopage
\pagenumbering{arabic}
\setcounter{section}{0}
\setcounter{page}{1}

\input{\path body}

%------------------------------- APPENDICES ----------------------------------

\sectionpaging
\appendix
\begin{center}
        \Large \bf APPENDICES \vspace{.5in}
        \normalsize
\end{center}
\addtocontents{toc}{\protect\vspace{5ex}}
\addtocontents{toc}{}
\addtocontents{toc}{\protect\begin{center}\protect\Large \protect\bf
        Appendices \protect\end{center}\protect\vspace{5ex}\protect\normalsize}
\addtocontents{toc}{}

\input{\path apprA}

\sectionpaging
\input{\path apprB}

\sectionpaging
\input{\path apprC}

%------------------------------- REFERENCES ----------------------------------

\sectionpaging
\input{\path bibl}

%------------------------------- END OF DOCUMENT -----------------------------

\end{document}
```

B.2 The Program Compilation Script

```
@echo off
rem  (31-Jul-1992)                                              (CJ_code.bat)
rem ##########################################################################
rem #                                                                        #
rem #       Copyright (C) 1992, California Institute of Technology          #
rem #     All rights reserved.  U. S. Government sponsorship under NASA     #
rem #                 Contract NAS7-918 is acknowledged.                    #
rem #                                                                        #
rem #                        Robert C. Tausworthe                            #
rem #                      Jet Propulsion Laboratory                         #
rem #                        4800 Oak Grove Drive                            #
rem #                      Pasadena, CA 91009-8099                           #
rem #                                                                        #
rem #                                                                        #
rem ##########################################################################

rem     ConJoin environment transformation for MAKE, and MAKE the ConJoin
rem     Program.

if exist CJ_code.mak goto ok
echo No file CJ_code.mak
goto fin

:ok
rem     set up the MAKE environment for locating include files, libraries,
rem     the MAKE program, and temporary file i/o:

set savepath=%path%
set path=c:\c\msc5\bin;%savepath%
set include=c:\c\ansi\h;c:\c\topc\h
set lib=c:\c\lib
set init=c:\c\msc5\bin
set tmp=c:\temp

rem     MAKE the ConJoin program

make CJ_code.mak

rem     restore the environment

set path=%savepath%
set savepath=
set lib=
set tmp=
set include=
set init=

:fin
```

B.3 The Program Compilation MAKE File

```
# (31-Jul-1992)                                                    (CJ_code.mak)
############################################################################
##                                                                         #
##          Copyright (C) 1992, California Institute of Technology         #
##       All rights reserved.  U. S. Government sponsorship under NASA     #
##                    Contract NAS7-918 is acknowledged.                   #
##                                                                         #
##                        Robert C. Tausworthe                             #
##                       Jet Propulsion Laboratory                         #
##                        4800 Oak Grove Drive                             #
##                       Pasadena, CA 91009-8099                           #
##                                                                         #
##                                                                         #
############################################################################

#                 ConJoin program construction instructions

ConJoin.exe: ConJoin.c CJ_code.mak
    cl /W3 /Ox ConJoin.c /F 1000 /link /NOE stopc.lib;
    del ConJoin.obj
```

The compiler is directed to print all warning messages and make the program internal stack size be 0x1000. The program is built using the small memory model, the compiler default. The automatic linking process searches the small-memory-model TOP-C library for the functions discussed in Section 8.

B.4 The Document Construction Script

```
@echo off
rem  (31-Jul-1992)                                                    (CJ_doc.bat)
rem ##########################################################################
rem #                                                                        #
rem #         Copyright (C) 1992, California Institute of Technology         #
rem #       All rights reserved.  U. S. Government sponsorship under NASA    #
rem #                   Contract NAS7-918 is acknowledged.                   #
rem #                                                                        #
rem #                        Robert C. Tausworthe                            #
rem #                       Jet Propulsion Laboratory                        #
rem #                        4800 Oak Grove Drive                            #
rem #                       Pasadena, CA 91009-8099                          #
rem #                                                                        #
rem #                                                                        #
rem ##########################################################################

rem     MAKE the ConJoin usage message and entire document.

make /I cj_usage.mak
make cj_doc.mak
```

Two separate MAKE invocations are made. The first, made under the /I option, causes MAKE to ignore an error return code from processors called by MAKE. This option is necessary, because CJ_usage.mak executes ConJoin with no command arguments, a condition that causes the usage message to print and the program to terminate with an error-indicating return code. The ConJoin output, in this case, is redirected into the CJ_usage.msg file for %access into Section 4.2 of this document.

The second MAKE invocation compiles and builds the document you are reading.

B.5 The Usage Message MAKE File

```
# (31-Jul-1992)                                                (CJ_usage.mak)
##############################################################################
##                                                                           #
##         Copyright (C) 1992, California Institute of Technology            #
##       All rights reserved.  U. S. Government sponsorship under NASA       #
##                   Contract NAS7-918 is acknowledged.                      #
##                                                                           #
##                         Robert C. Tausworthe                              #
##                       Jet Propulsion Laboratory                           #
##                         4800 Oak Grove Drive                              #
##                       Pasadena, CA 91009-8099                             #
##                                                                           #
##                                                                           #
##############################################################################

#  This is a separate MAKE file because it is executed with the /I
#  option to ignore exit codes, because ConJoin returns an error code on
#  termination.

CJ_usage.msg: Conjoin.exe
    ConJoin >CJ_usage.msg
```

B.6 The Document MAKE File

```
# (31-Jul-1992)                                                (CJ_doc.mak)
##############################################################################
##                                                                          #
##          Copyright (C) 1992, California Institute of Technology          #
##       All rights reserved.  U. S. Government sponsorship under NASA      #
##                   Contract NAS7-918 is acknowledged.                     #
##                                                                          #
##                           Robert C. Tausworthe                           #
##                        Jet Propulsion Laboratory                         #
##                           4800 Oak Grove Drive                           #
##                         Pasadena, CA 91009-8099                          #
##                                                                          #
##                                                                          #
##############################################################################

# Directory of TOP-C Functions
topc=\c\topc\c\ #


CJ_prog.cal: ConJoin.c CJ_prog.cal
    calltree /c CJ_prog.cal /b CJ_prog.by ConJoin.c

CJ_prog.by: ConJoin.c CJ_prog.by
    calltree /c CJ_prog.cal /b CJ_prog.by ConJoin.c

CJ_prog.siz: ConJoin.c
    flines ConJoin.c >CJ_prog.siz

CJ_appxA.tex: CJ_AppxA.cjn ConJoin.c
    ConJoin -a CJ_appxA

CJ_appxB.tex: CJ_AppxB.cjn CJ_reprt.tex cj_code.bat cj_doc.bat \
        cj_code.mak cj_doc.mak
    ConJoin -a CJ_appxB

CJ_appxC.tex: CJ_appxC.cjn CJ_prog.cal CJ_prog.by
    ConJoin -a CJ_appxC

CJ_body.tex: CJ_body.CJn ConJoin.c CJ_style.tex CJ_usage.msg \
        CJ_fig1.tex CJ_prog.siz $(topc)stratrim.c $(topc)strdup.c \
        $(topc)strfnb.c $(topc)strlwr.c $(topc)strtcpy.c $(topc)strtrim.c
    ConJoin -a CJ_body

CJ_reprt.dvi: CJ_reprt.tex CJ_style.tex CJ_abstr.tex CJ_ackno.tex \
        CJ_credt.tex CJ_body.tex CJ_appxA.tex CJ_appxB.tex \
        CJ_appxC.tex CJ_bibl.tex CJ_doc.mak
    call latex CJ_reprt
    call dvi CJ_reprt
```

The `calltree` program is a utility supplied with Microsoft C. It prints out call trees (structure diagrams) and reference trees (called-by diagrams). These appear in Appendix C.

The **flines** utility scans the named file and writes the number of lines in that file normally to the console, but here to a size file accessed by **CJ_body.CJn**.

The *ConJoin* program creates the **.TeX** versions of Appendices A, B, and C and the text body; the **-a** option suppresses printing of the JPL/Caltech copyright notice and program announcement.

This report was then composed by the LaTeX system, which is set up and executed **called** as a batch command, **latex**.

This report was then printed by **calling** the batch command **dvi**, which converts the device-independent output of TeX into the typeset form that you are reading now.

C ConJoin FUNCTIONAL STRUCTURE

C.1 Call Tree

The following listing denotes the invoked function structure of **ConJoin** in depth-first order. Multiple branches of the same subtree are not shown. Function names followed by '?' are external to **ConJoin**, and are either in the standard ANSI library or the TOP-C library. Although not functions, some defined terms, such as **NIL**, **NOT**, **AND**, and other spurious entries may appear in the listing because of a flaw in the Microsoft **calltree** utility used to generate the tree.

```
NIL
FGETSIZE
strdup
strtrim
fgets
main
|   initialization
|   |   command_line
|   |   |   strlwr?
|   |   |   sprintf?
|   |   |   strlen?
|   |   |   strcpy?
|   |   |   strcat?
|   |   announce
|   |   |   printf?
|   |   usage
|   |   |   printf?
|   |   error_message
|   |   |   printf?
|   |   |   fprintf?
|   |   file_defaults
|   |   |   NOT?
|   |   |   strchr?
|   |   |   strcat?
|   |   |   strtcpy?
|   |   strcmp?
|   |   strcpy?
|   |   strchr?
|   |   fopen?
|   |   FGETSIZE...
|   |   fclose?
|   open_io_files
|   |   file_open
|   |   |   NOT?
|   |   |   fopen?
|   |   |   strcpy?
|   |   |   strncat?
|   |   |   strcat?
|   |   |   strerror?
|   |   |   error_message...
|   |   fopen?
```

PRECEDING PAGE BLANK NOT FILMED

126 INTENTIONALLY BLANK

```
|   |   fclose?
|   |   strchr?
|   |   strcpy?
|   |   remove?
|   |   rename?
|   timestamp
|   |   strlen?
|   |   strnset?
|   |   time?
|   |   localtime?
|   |   stratrim?
|   |   strcpy?
|   |   asctime?
|   |   right_fill
|   |   |   strnset?
|   |   |   strlen?
|   |   fprintf?
|   ConJoin_files
|   |   fgets...
|   |   strfnb?
|   |   fprintf?
|   |   strstr?
|   |   strcpy?
|   |   directive
|   |   |   strcpy?
|   |   |   strstr?
|   |   |   strext
|   |   |   |   AND?
|   |   |   |   strstr?
|   |   |   |   strtcpy?
|   |   |   |   strcpy?
|   |   |   |   stratrim?
|   |   |   strlen?
|   |   |   atoi?
|   |   |   strnset?
|   |   |   strfnb?
|   |   |   strdup...
|   |   |   strlwr?
|   |   |   stratrim?
|   |   |   strcmp?
|   |   |   error_message...
|   |   |   sprintf?
|   |   |   strinsert?
|   |   |   AND?
|   |   |   strcat?
|   |   access
|   |   |   strext...
|   |   |   match_parameters
|   |   |   |   NOT?
|   |   |   |   strstr?
|   |   |   |   strcpy?
|   |   |   |   error_message...
|   |   |   |   stratrim?
|   |   |   |   strtcpy?
|   |   |   |   strfnb?
|   |   |   |   strlen?
|   |   |   |   AND?
|   |   |   |   access_condition
|   |   |   |   |   stratrim?
|   |   |   |   |   strstr?
|   |   |   |   |   atoi?
|   |   |   |   |   strlen?
|   |   |   |   |   strchr?
|   |   |   |   |   OR?
|   |   |   |   |   isdigit?
|   |   |   |   |   strdup...
|   |   |   |   |   strtrim...
|   |   |   |   |   strfnb?
|   |   |   NOT?
|   |   |   open_access
|   |   |   |   strcpy?
|   |   |   |   NOT?
|   |   |   |   fopen?
|   |   |   |   strcat?
|   |   |   error_message...
|   |   |   printf?
|   |   |   fprintf?
|   |   |   scan_to_bgn_match
|   |   |   |   fgetpos?
|   |   |   |   fgets...
|   |   |   |   strstr?
|   |   |   |   fsetpos?
|   |   |   |   error_message...
|   |   |   copy_to_end_match
|   |   |   |   fgetstr
|   |   |   |   |   NOT?
|   |   |   |   |   fgets...
|   |   |   |   |   isspace?
|   |   |   |   |   ORisprint?
|   |   |   |   |   strcat?
|   |   |   |   |   strtrim...
|   |   |   |   NOT?
|   |   |   |   strdup...
|   |   |   |   error_message...
|   |   |   |   free?
|   |   |   |   putline
|   |   |   |   |   AND?
|   |   |   |   |   strchr?
|   |   |   |   |   min?
|   |   |   |   |   fprintf?
|   |   |   |   strstr?
|   |   |   free?
|   |   |   fclose?
|   |   strlen?
|   fclose?
|   free?
|   printf?
|   fopen?
|   fprintf?
|   access_lines?
```

C.2 Reference List

The following list displays the invoked-by structure of functions within the ConJoin program. As in the previous section of this appendix, NIL appears at the whim of the reference-tree-producing tool.

```
     ConJoin_files: main

          FGETSIZE: initialization

               NIL:

            access: ConJoin_files

  access_condition: match_parameters

          announce: initialization

      command_line: initialization

 copy_to_end_match: access

         directive: ConJoin_files

     error_message: access        copy_to_end_match directive
                    file_open     initialization match_parameters
                    scan_to_bgn_match

             fgets: ConJoin_files fgetstr       scan_to_bgn_match

           fgetstr: copy_to_end_match

     file_defaults: initialization

         file_open: open_io_files

    initialization: main

              main:

  match_parameters: access

       open_access: access

     open_io_files: main

           putline: copy_to_end_match

        right_fill: timestamp

 scan_to_bgn_match: access

            strdup: access_condition copy_to_end_match directive

            strext: access        directive
  strtrim: access_condition fgetstr
timestamp: main
    usage: initialization
```

D REFERENCES

[1] Spear, Barbara, *How to Document Your Software*, TAB Books, Inc., Blue Ridge Summit, PA, 1984.

[2] Jung, Karl, *Psychological Types*, London Press, 1923.

[3] Martin, James, and McClure, Carma, *Diagramming Techniques for Analysts and Programmers*, Prentice-Hall, Inc, Englewood Cliffs, NJ, 1985.

[4] Lamport, Leslie, *LaTeX: A Document Preparation System* User's Guide and Reference Manual, Addison-Wesley Publishing Company, Reading, MA, 1986.

[5] Press, William H., et al., *Numerical Recipes*, Cambridge University Press, Cambridge, England, 1986.

[6] Goldfarb, C. F., *The SGML Handbook*, Oxford University Press, 1990.

[7] *Hypermedia/Time-based Document Structuring Language* ISO/IEC Draft International Standard 10744, International Organization for Standardization and International Electrotechnical Commission.

[8] Newcomb, S. R., et al., "The HyTime Hypermedia/Time-Based Document Structuring Language," *Communications of the ACM*, Vol. 34, No. 11, November 1991, pp. 67-83.

[9] International Standard 8879, "Information processing—Text and office systems—Standard Generalized Markup Language (SGML)," International Orgainization for Standards, Reference No. 8879-1986 (E).

[10] Haan, Bernard J., et al., "IRIS Hypermedia Services," *Communications of the ACM*, Vol. 35, No. 1, January 1992, pp. 36-51.

[11] Knuth, Donald E., "The WEB system of structured documentation," Computer Science Report 980, Stanford University, Palo Alto, CA, September 1983.

[12] Knuth, Donald E., *The TeXbook*, Addison-Wesley Publishing Company, Reading, MA, 1984.

[13] Knuth, Donald E., "Literate Programming," *The Computer Journal*, Vol. 27, No. 2, May, 1984, pp. 97–111.

[14] Cordes, David, and Brown, Marcus, "The Literate Programming Paradigm," *IEEE Computer*, Vol. 24, No. 6, June, 1991, pp. 52–62.

[15] Hyman, Marco C., "Literate C++," *Computer Language*, Vol. 7, No. 7, June 1991, pp. 67–79.

[16] Levy, S., "**WEB** Adapted to C—Another Approach," *TUGboat*, Vol. 8, No. 1, April, 1987, pp. 12–14.

[17] Thimbleby, Harold, "Experiences in 'literate programming' using **CWEB**," *The Computer Journal*, Vol. 29, 1986, pp. 201–211.

[18] Sewell, Wayne, *Weaving a Program: Literate Programming in* **WEB**, Van Nostrand Reinhold, New York, 1989.

[19] *Webster's Ninth New Collegiate Dictionary*, Merriam-Webster Inc., Springfield, MA, 1983.

[20] Tausworthe, Robert C., "A General Software Reliability Process Simulation Technique," Publication 91-7, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, April 1, 1991.

[21] Meyer, John, *XEROX Ventura Publisher Edition Reference Guide*, Xerox Corp., 1987.

[22] Elsayed, E. A., Taguchi, G., and Tsiang, T., *Quality Engineering in Production Systems*, McGraw-Hill Book Co., NY, 1988.

[23] Plum, Thomas, *C Programming Standards and Guidelines*, Plum Hall Inc., Cardiff, NJ, 1982.

[24] Thomas, Edward J., et al., "A Bibliography of Programming Style," *SIGPLAN Notices*, Vol. 25, No. 2, 1990, pp. 7–16.

[25] Tausworthe, Robert C., *Standardized Development of Computer Software*, Volume 1: Methods, Volume 2: Standards, Prentice-Hall Inc., Englewood Cliffs, NJ, 1977 and 1979.

[26] Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1978.

[27] *American National Standard for Information Systems—Programming Language C*, X3.159-1989, Draft X3J11/88-158, American National Standards Institute, December 7, 1988.

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No. 92-12	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle Conjunctive Programming--An Integrative Approach to Software System Synthesis	5. Report Date August1, 1992
	6. Performing Organization Code
7. Author(s) Robert C. Tausworthe	8. Performing Organization Report No.
9. Performing Organization Name and Address JET PROPULSION LABORATORY California Institute of Technology 4800 Oak Grove Drive Pasadena, California 91109	10. Work Unit No.
	11. Contract or Grant No. NAS7-918
	13. Type of Report and Period Covered JPL Publication
12. Sponsoring Agency Name and Address NATIONAL AERONAUTICS AND SPACE ADMINISTRATION Washington, D.C. 20546	14. Sponsoring Agency Code BK-506-59-11-01-00 (Task: RE156)

15. Supplementary Notes

16. Abstract

This report introduces a technique of software documentation called conjunctive programming and discusses its role in the development and maintenance of software systems. The report also describes the ConJoin tool, an adjunct to assist practitioners. Aimed at supporting software reuse while conforming with conventional development practices, conjunctive programming is defined as the extraction, integration, and embellishment of pertinent information obtained directly from an existing database of software artifacts, such as specifications, source code, configuration data, link-edit scripts, utility files, and other relevant information, into a product that achieves desired levels of detail, content, and production quality. Conjunctive programs typically include automatically generated tables of contents, indexes, cross references, bibliographic citations, tables, and figures (including graphics and illustrations). This report presents an example of conjunctive programming by documenting the use and implementation of the ConJoin program.

17. Key Words (Selected by Author(s)) Documentation and Information Technology Mathematical and Computer Sciences Computer Programming and Software	18. Distribution Statement Unlimited--Unclassified

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 134	22. Price

JPL 0184 R 9/83