MEASURING COMPLEXITY AND STABILITY OF WEB PROGRAMS

By

LISA MIN-YI CHEN SMITH

Bachelor of Science

in Arts and Sciences

University of Kentucky

Lexington, Kentucky

1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1990

MEASURING COMPLEXITY AND STABILITY OF

WEB PROGRAMS

Thesis Approved:

M. Samadzadeh-H.

Thesis Advisor

D. E. Hedrick

William David Miller

Norman N. Durham

Dean of the Graduate College

PREFACE

Software maintenance engineers spend at least half of their time trying to understand the system they are to modify. This is due partially to the fact that often the only documentation available is the source code itself. The *literate programming paradigm* provides the incentive and the capability to produce high quality documentation and code simultaneously. The goal is to create "works of literature" which have all the extras (table of contents, cross references, and indices) to help readers to comprehend the programs quickly and thoroughly. The purpose of this thesis is to explore the similarities and differences in measurements of complexity and stability of literate programs compared to those of traditionally developed code.

My sincere appreciation goes to my major advisor, Dr. Mansur H. Samadzadeh, who has been patient, enthusiastic, and full of encouragement from day one. Without his ideas, guidance, and library of books, I would never have come close to finishing this thesis.

I would also like to thank Drs. George E. Hedrick and W. David Miller for their comments and suggestions while serving on my committee.

In addition, special thanks go to Dr. S. Bart Childs of Texas A&M University for advancing my research of literate programming by leaps and bounds.

My appreciation goes to Mr. Steve Koinm for helping me to get started on using TeX, **TANGLE**, and **WEAVE**; in addition to responding to my cries of help when the laser printer was not cooperating.

I would like to thank my parents, Boris and Linda Chen, for their generous support, and my sister Audrey for listening to me. I would like to express my gratitude to James and Beverly Smith and Mrs. Idella Smith for their encouragement. Most of all, I would like to thank Gary for being a wonderful husband and for helping me to survive Graduate School.

A Note on Format: Appendix B (pp. 60–106) and C (pp. 107–118) are not in strict compliance with the OSU Graduate School Thesis Format requirements regarding the margins and the numbering of pages. This deviation is due to the fact that the format of those appendices, as well as their contents, is part of the programming environment being promoted in this thesis.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER I

INTRODUCTION

The software crisis is upon us. One of the major problems we face is that maintenance activities (correcting errors, adapting a system to a new environment, or adding enhancements to fulfill new requirements or improve performance) consume half of all resources allocated to software development [PaZv83]. Before a system can be modified, it must be understood by the software engineer(s) performing the task(s). It is disturbing that at least half of their time is spent trying to understand what the alien code does (or is supposed to do!) [PaZv83]. This may be due, in part, to the fact that quite often the only information available to the maintainer is the source code of the program itself. Thus, quality of the documentation will play a major part in how quickly and completely a piece of software will be understood. The *literate programming paradigm* introduced publically by Knuth in 1984, provides the incentive and the capability for producing such documentation.

CHAPTER II

LITERATE PROGRAMMING

2.1 Background

The role of documentation is the crucial difference between traditional programs and literate programs. Traditional programs are written for a computer to execute, with comments added to show the meaning of some parts of the code. In contrast, literate programs contain documentation to explain what the program does in a manner which facilitates understandability and readability by a human audience. Documentation is no longer secondary, but is at least of equal importance, if not more important, than the code itself. This investment of documenting during development should more than pay for itself during program maintenance [Thim86].

2.1.1 Definition

Lins [Lin89a] sums the concept in this manner:

literate programming = structured programming + structured documentation

Thus, a literate program contains both source code and its documentation. The two may be presented in any order the author believes will enhance his ability to explain the program (generally, the order in which it is written). Using a utility program, source code can be extracted from a literate program, be compiled, and executed. Using a different utility, a typeset document containing all source code and documentation can be created. It is crucial that the program executed and the document produced be created from exactly the same input file. In addition, niceties available in a traditional work of literature, such as a table of contents, cross references, and an index should also be generated automatically. These are the features of the literate programming paradigm [VWyk90, VWLT89].

2.1.2 Advantages and Limitations

Literate programming provides many advantages over traditional program development. The discipline of simultaneously documenting programs while developing the code leads to significantly better programs and documentation [BeKM86, ReSk89]. Since the explanation of code and its implementation are so tightly coupled, it becomes very difficult to gloss over the inscrutable parts, thus helping both the author to explain the code and his readers to understand it [BeKM86]. In addition, since the program is hidden in its documentation, it is impossible to modify the code without changing the documentation at the same time [Leca85]. Mixing general descriptions with precise code segments is much more powerful than thinly interspersed comments found in traditional programs [ReSk89]. Since commentary becomes more prominent, even better documentation is encouraged as any omissions are now readily apparent. Also, since the compilation order of the code no longer dictates how the program is designed and presented, the resulting program is much more comprehensible and thus will be more maintainable for the future.

Thimbleby [VWLT89] states: "How literate programming is done, and how easily it can be done and redone, changes the way one programs. It provides new incentives. There is an incentive to make code and documentation consistent (by developing code and documentation concurrently). There is an incentive to explain and hence understand what you are doing. And by making a program look so nice, it gives an incentive to publicize the program and suffer its public review!"

On the other hand, literate programming does have some limitations which may limit its widespread use. To write a literate program, the author must know several languages: a high level programming language, a text formatting language, a literate programming specification language, and English. Since the program can generate three types of syntax errors, plus algorithmic errors, some sophistication and patience is needed to debug the program [Knut84]. Quality of documentation is not guaranteed; it is very person-dependent [ReSk89]. Knuth comments that literate programming may only be for those who "like to write and explain what they are doing" [Knut84].

In addition, some limitations of literate programming can be blamed on its infancy. Almost all literate programs published so far have been written by Knuth himself [BeKn86, BeKM86, Knu86a, Knu86b]. By inspection, all of them appear to have been programmed from scratch. For literate programming to become widespread, development with reuse in mind must be considered [BeKM86]. Another improvement which is needed is the ability to add diagrams to the documentation, for illustrating difficult data structures and interrelationships among various program components. The lack of software tools to support literate programming is another limitation [Lin89a]. For example, the advantage of being able to present source code and documentation in any manner seen to be more comprehensible, causes the traditional graph structure of the total program hardly to be visible. This is the one of the best tools for measuring program complexity [Leca85] and would be useful to have.

2.2 The WEB System

2.2.1 TeX, TANGLE, and WEAVE

Knuth developed the literate programming methodology when developing the second version of his software system for typesetting, TeX. Originally, TeX was written in SAIL, a language not widely available. Wanting to make TeX more portable, Knuth developed a system called DOC, for structured documentation. In 1981, he replaced DOC with WEB and it has been his programming language of choice ever since [Knut89].

Knuth's WEB system embodies the ideas of literate programming in the WEB language and its associated programs. It is a combination of a document formatting language and a programming language. For his prototype, Knuth chose TeX as the document formatting language and Pascal as the programming language [Knut84].

Knuth wrote the literate programs, TeX82 and *Metafont*, using his WEB system of structured documentation. WEB consists of the two system routines TANGLE and WEAVE. TANGLE takes a WEB program, extracts and rearranges the interleaved code from the documentation and rearranges it to produce a syntactically correct Pascal program ready to be compiled. WEAVE also takes the WEB program as input, but produces a document containing pretty-printed source code and documentation that is ready to be TeXed.

2.2.2 Code Sections

A WEB program consists of WEB commands, TeX commands, and Pascal code. Each program consists of a series of numbered code sections. A code section can have at most three parts which must appear in the following order, but any part may be empty.

I. *informal commentary* written in TeX, explaining what the current section does;

II. *macro definitions*, which are abbreviations for Pascal constructions that make the code more readable and portable; and

III. *pascal code* to be extracted by `TANGLE`.

Every code section is either named or unnamed. A named section begins with its name followed by its Pascal code (also called its replacement text). When a named section appears in another section, the corresponding replacement text should be substituted because a named section serves as a placeholder for its replacement text. If more than one section has the same name, the latter section's Pascal code is appended to the former section's replacement text and relative order is maintained. In addition, a section may also be categorized as a major section. In the `WEAVE`d (woven?) output, all major sections appear in the table of contents and start on a new page.

2.2.3 Macros

There are three types of macros that can be defined in a `WEB` program. Use of macros is encouraged for promoting portability and readability of programs.

- a *numeric macro* associates numbers with identifiers and allows `TANGLE` to do simple arithmetic;
- a *simple macro* causes `TANGLE` to replace an identifier with Pascal text; and
- a *parametric macro* causes an identifier to be replaced by Pascal text, and all occurrences of the parameter (denoted by #) to be replaced by an argument.

2.2.4 Indexing

The WEB system provides several mechanisms for cross referencing. A *table of contents* is provided with names of all major sections and their page numbers. The *index* lists all identifiers that appear in the program, along with the section in which each one was declared (underlined) and all other sections where they were used. Finally, an alphabetized *list of all named sections* is provided, with a list of all sections where each named section is used (see Appendix B or C for an example).

2.2.5 CHange Files

A WEB program may have a CHange file associated with it. A CHange file consists of zero or more changes, where each change contains a block of text from the original WEB program (to be modified), and a block of text which is to replace the original text. That is, when WEAVE and TANGLE are run, they replace the original block of text with the new changed block of text, for each change encountered. The CHange file is very useful in program maintenance when customizing system-dependent changes or adding enhancements to the code. To create a new validated version of a WEB program, it may be desired to merge the CHange file with the original WEB file after a major bug has been fixed and fully tested.

2.3 Other Systems/Research

Knuth's WEB System has been used more as a model for other literate programming systems than for development of literate programs. Some of the other literate programming systems use a different programming language or text formatting language, while using most features of WEB, along with some new ones [VWyk90]. The following subsections briefly describe some of the literate programming systems that have been modeled after the `WEB` System.

2.3.1 `Cweb`

The first such system was implemented by Thimbleby in 1986, using the programming language C and the text formatting language **troff**. He named his `UNIX` version **`cweb`**. Several differences exist between **`cweb`** and the original `WEB`, including the following: (1) in **`cweb`**, macro bodies can be placed in named files which can be *included* in a program, and thus be reused; (2) **`cweb`** does not pretty print the C code like `WEB` does the Pascal code; (3) **`cweb`** can produce output for line printers, but `WEB` can't; and (4) if a **`cweb`** author updates the documentation of his/her literate program, without touching any code, the code (i.e., the `TANGLE`d part) need not be recompiled. During the actual implementation of **`cweb`**, Thimbleby estimated he spent 95% of the time in text formatting (**troff**) related issues. His observations for future work include the need for development of language-independent literate programming notation, interactive editors, or integrating literate programming systems into new languages [Thim86].

2.3.2 Spider `WEB`

Ramsey recently published his work on a language-independent version of `WEB` [VWRa89, Ram89a, Ram89b, Sewe89]. His program, called `SPIDER`, generates a variant of Knuth's `WEB` System by combining TeX with a programming language X of your choice (instead of Pascal). The description of the desired programming language X is combined with language-independent master files of `TANGLE` and `WEAVE` to produce C code for `XTANGLE` and `XWEAVE` [VWRa89]. Ramsey has generated `WEB` systems for C, AWK, SSL, and Ada, among other languages.

2.3.3 A Literate Programming Environment

In other research on literate programming, a prototype interactive WEB browser using a hypertext structure has been implemented by Brown [Bro88a, Bro88b, BrCh90, BrCh89]. This WEB tool allows the user to view the program in different ways, including one section at a time or as a tree of sections. Browsing is accomplished by clicking the mouse at certain hot spots on the screen. Sections can be brought up from either the index or a named sections list. When a code section is displayed, it resembles the TeXed output rather than the WEB source code (no WEB commands appear). The addition of editing capabilities should enhance the browser's usefulness in a Literate Programming Environment [Sewe89].

Work in progress of other literate programming environments include systems for FORTRAN [AvOp90] and Smalltalk [ReSk89]. See Appendix A for an annotated bibliography of literate programming.

CHAPTER III

SOFTWARE METRICS

Conte, Dunsmore, and Shen define software metrics as measures which can be used to quantify software such that it can be classified, compared, and analyzed mathematically [CoDS86]. Software metrics can be divided into various categories. Yau and Collofello distinguish between metrics of the design phase of the software life cycle vs. source code metrics which are gathered during the coding phase [YaCo85]. There are a large number of software metrics which measure software complexity and some that have been developed to monitor the stability of software [YaCo80]. Some of the classic and recently developed metrics of the two categories are outlined below.

3.1 Complexity

Four well-known and popular complexity measures are described below. The two design-phase measures include McCabe and Butler's recent design complexity metrics based on cyclomatic complexity, and Henry and Kafura's information flow complexity metrics based on fan-in and fan-out. The source code measures to be covered are Halstead's Software Science metrics and McCabe's cyclomatic number.

Measurements which can be collected before the coding phase of the software life cycle are known as macro-level metrics [KaHe81]. These metrics typically focus on the relationship between system components (procedures or modules). The ability to discover software design flaws in an early phase of the software development process is a major advantage of these measures.

3.1.1 Design Complexity

McCabe and Butler recently published their work [McBu89] using cyclomatic complexity [McCa76] to measure design complexity. They define the following design metrics: module design complexity, design complexity, and integration complexity. The three design metrics, which are described below, are essentially based on the concept of cylcomatic complexity. Cyclomatic complexity, v(G), is a measure of program control flow complexity: the number of basic paths through a flowgraph G. The easiest way to calculate v(G) is to count decision statements (predicates) in the program.

$$v(G) = \text{number of decision statements} + 1 \tag{1}$$

What follows in this section is a detailed description of the three design metrics.

I. *module design complexity* **iv**

Each individual module in a design has its own flowgraph G. A module's primary control structure of calling subordinate modules can be determined after four reduction rules are applied to the flowgraph.

> The cyclomatic complexity of the reduced flowgraph is the module design complexity, $iv(G)$, of the original flowgraph G. (2)

The notation iv(G) is derived in an attempt to be indicative of its purpose, i.e., calculating *i*ndividual cyclomatic complexity, *v(G)*.

II. *design complexity* S_0

The primary design instrument in this phase of the development cycle is the structure chart or hierarchy tree. A structure chart defines the manner in which modules work together, but not how each individual module works. The *design complexity*, S_0, of the structure chart of a module M is defined below.

$$S_0 = \sum_{i \in D} \mathbf{iv}(\mathbf{G_i}) \tag{3}$$

where $D = \mathrm{M} \cup \{$ descendents of M $\}$ and M is a module.

III. *integration complexity* S_1

The last design metric McCabe and Butler define is a measure of the number of integration tests required to test the overall design.

$$S_1 = S_0 - n + 1 \tag{4}$$

where n is the number of modules.

The following algorithm for computing design and integration complexity is adapted from McCabe and Butler [McBu89].

Algorithm for Computing Design and Integration Complexity

Input: Program code or design psuedocode.

Output: $iv(x)$ and $S_0(x)$, $\forall$ module x; and S_1.

Method:

step 1. $\forall$ module x, construct a module flowgraph. Each white dot should correspond to a block of code where the flow is sequential, a black dot should correspond to a subordinate module call, and arcs correspond to branches in the code.

step 2. Construct a structure chart or design tree for the program. This defines how the modules of the program work together. Show a *call* from a superordinate to a subordinate module with a black dot.

step 3. $\forall$ module x, apply the following four reduction rules to produce its primary control structure for calling subordinate modules — (1) sequential black dot: a call to a subordinate module cannot be reduced; (2) sequential white dot: a sequential node can be reduced to a single edge; (3) repetitive white dots: a logical repetition without a black dot can be reduced to a single node; and (4) conditional while dots: a logical decision with two paths without a black dot can be reduced to one path.

step 4. $\forall$ module x, compute *individual module design complexity*, $iv(x)$, from the reduced flowgraph $iv(x) =$ number of conditions (predicates) + 1.

step 5. $\forall$ module x, compute *design complexity*, $S_0(x)$, using one of the following two options: (a) If module x's design is a pure tree (it has no common modules), S_0 is upwardly additive: $S_0(x) = iv(x) + \sum_{i \in D} S_0(G_i)$, where $D = \{$ descendents of x $\}$; (b) If module x's design is not a pure tree (has common modules), S_0 is nonadditive. (This case happens more often.) $S_0(x) = \sum_{i \in D} iv(G_i)$, where $D =$ x $\cup$ $\{$ descendents of x $\}$.

step 6. Calculate integration complexity, S_1, for the desired modules. $S_1 = S_0 - n + 1$, where $n =$ number of modules in the design.

Note that a design where $S_0 = n$ always behaves in the same way. There are no conditional calls to subordinate modules. Therefore, $iv(G) = 1$ for each module in the design.

3.1.2 Fan-in/Fan-Out

Henry and Kafura defined four information flow complexity measures which can be determined during the design phase (as well as the coding phase) of the software life cycle [HeKa81, KaHe81]. They provide several definitions of various types of information flow on which these metrics are based.

Def 1. There is a *global flow* of information from module M_1 to module M_2 through a global data structure D if M_1 deposits information into D and M_2 retrieves information from D.

Def 2. There is a *local flow* of information from module M_1 to module M_2 if one or more of the following conditions hold: (a) if M_1 calls M_2; (b) if M_2 calls M_1 and M_1 returns a value to M_2, which M_2 subsequently utilizes; or (c) if M_3 calls both M_1 and M_2 passing an output value from M_1 to M_2.

Def 3. The *fan-in* of a procedure A is the number of local flows into procedure A plus the number of data structures from which procedure A retrieves information.

Def 4. The *fan-out* of a procedure A is the number of local flows from procedure A plus the number of data structures which procedure A updates.

The measures calculated using these concepts are the complexity value of a procedure, complexity of a module, the number of global flows of a module, and strength of connections between two modules.

I. *complexity value of a procedure*

The *complexity value of a procedure* is based on the bulk complexity of the procedure code, *length*, and the complexity of the procedure's connections to its environment, $fan\text{-}in * fan\text{-}out$. The justification offered for the power of two is Brook's law of programmer interaction [Broo75] and Belady's formula for system partitioning [BeEv79].

$$\text{complexity value of a procedure} = length * (fan - in * fan - out)^2 \qquad (5)$$

II. *complexity of a module*

According to Henry and Kafura, a *module* with respect to a data structure D consists of those procedures which either directly update D or directly retrieve information from D. Thus, module complexity is calculated using procedure complexities.

$$\text{complexity of a module} = \sum_{p \in M} \text{complexity of a procedure } p \tag{6}$$

where M is a module.

III. *number of global flows of a module*

The *number of global flows of a module* is an indicator of the number of procedures in each module and which procedures *read-only*, *write-only*, or *read-write* to the data structures.

$$\begin{aligned}&\text{number of global flows of a module } =\\ &(write * read) + (write * readwrite) +\\ &(readwrite * read) + (readwrite * (readwrite - 1))\end{aligned} \tag{7}$$

IV. *strength of connections from module M_1 to module M_2*

Interfaces between modules show how components are connected to form the overall system. Minimizing connection among modules can be used as a design goal.

$$\begin{aligned}&\text{strength of connections from module } M_1 \text{ to module } M_2 =\\ &(\text{the no. of proc. exporting info. from module } M_1 +\\ &\text{the no. of proc. importing info. into module } M_2) *\\ &\text{the no. of info. paths.}\end{aligned} \tag{8}$$

Source code metrics focus on individual components because details of the internal modules and procedures are required. Another name for these measures are microlevel metrics [KaHe81]. The following two subsections describe two source code metrics.

3.1.3 Software Science

Halstead developed the Software Science family of measures which are calculated from four basic counts [Hals77, Hals79, RaMe88, BaZw80].

- η_1 = the number of unique operators
- η_2 = the number of unique operands
- N_1 = the total number of occurrences of all operators
- N_2 = the total number of occurrences of all operands

An operator is defined to be either a built-in function, a symbol or group of symbols that produce an action [RaMe88]. An operand can be a constant or a variable.

Some of Halstead's metrics are briefly described here.

I. *length* of a program in tokens N

$$N = N_1 + N_2 \tag{9}$$

II. *volume* of a program in bits V

The *volume* of a program is the fewest number of binary digits or bits with which the program can be represented.

$$V = N * log_2(\eta_1 + \eta_2) \tag{10}$$

III. *effort* spent developing a program E

The *effort* is the total number of elementary mental discriminations needed to write a program. The approximation of E, $\hat{E}$, does not need V^* (potential volume) for its calculation.

$$E = \frac{V^2}{V*} \tag{11}$$

$$\hat{E} = V * [\frac{(\eta_1 * N_2)}{(2 * \eta_2)}] \tag{12}$$

IV. *time* spent developing a program T

The *time* is *effort* converted to units of time based on the Stroud number, $S = 18$ e.m.d. per second.

$$T = \frac{E}{S} \tag{13}$$

It should be noted here that some of Halstead's metrics (time in particular) have been criticized because of the use of some results from experimental psychology. However, a large body of literature on validation across several programming languages, provides empirical support for most of Halstead's metrics.

3.1.4 Cyclomatic Complexity

McCabe's [McCa76] cyclomatic complexity metric measures program control flow complexity [RaMe88]. Its firm analytical basis ensures that it will be applied in measuring the complexity of graphs in a wide range of fields involving graph theory. There are several ways of calculating the cyclomatic complexity, three are mentioned below.

I. *cyclomatic number*, $v(G)$, of a graph G

$$v(G) = e - n + 2p \tag{14}$$

where e = number of edges, n = number of vertices, and p = number of connected components.

II. *cyclomatic complexity*, v, of a structured program

$$v = \pi + 1 \tag{15}$$

where π = number of conditions (predicates) in a program.

III. *cyclomatic complexity*, r, of a plane or planar control graph

$$r = e - n + 2 \tag{16}$$

where r = number of regions, e = number of edges, and n = number of vertices.

3.2 Stability

The following sections contain the description of some measures for determining the stability of a program in the design and coding phases which have been developed by Yau and Collofello [YaCo80, YaCo85].

3.2.1 Design Stability

Yau and Collofello define *design stability* as "the quality attribute indicating the resistance to the potential ripple effect which a program developed from the design would have when it is modified" [YaCo85]. Yau and Collofello's design stability measures are based on the use of data abstraction and information hiding, which profoundly affect the maintainability of a program. The lack of adequate data abstraction and information hiding in a design can result in modules possessing many assumptions about other modules in the design and/or its execution environment. During program maintenance, if changes are made which affect these assumptions, a ripple effect may occur throughout the program requiring additional costly changes.

The calculation of the design stability measures is restricted to the examination of the assumptions concerning *module interfaces* in the program. The *design stability of a module*, DS, is calculated as the reciprocal of the potential ripple effect as a consequence of modifying the module. The *potential ripple effect of a module*, $DLRE$, is defined as the total number of assumptions made by other modules, which either (a) invoke the module whose stability is being measured, (b) share global data or files with the module, or (c) are invoked by the module. This implies that modules with poor design stability are likely to affect many assumptions made by other modules, and consequently can produce a large ripple effect if modified. Finally, the *design stability of a program*, PDS, is calculated as the reciprocal of the total potential ripple effect of all its modules.

Some basic definitions are discussed below.

Module Interface. A module's interface is defined to consist of the module's *passed parameters, global variables,* and *shared files.* To be more precise, each of these objects must then be examined to see if they are composed of other identifiable entities. This decomposition into minimal entities is used to count *assumptions.* For example, a record can be decomposed into its respective fields and other structured data types can be decomposed into their respective basic types. Thus, if a record data type is part of a module's interface and that record consists of a character, an integer, and a real number, then the three minimal entities are the character, the integer, and the real number.

Assumptions. To standardize and simplify the recording of assumptions made by each module about its minimal interface entities, two categories of assumptions are utilized. The first type of assumptions concerns the *basic type* of the entity such as integer, real, Boolean, character, etc. This assumption is always recorded and can be checked automatically by a compiler. The second type of assumptions concerns the *value* of the basic entity and is recorded if the module has any assumptions about the values which the minimal entity may assume.

Counting Assumptions. Each minimal entity in an interface can contribute a maximum of one assumption to each category. The parameters, global variables, and shared files should also contribute to this assumption count. In general, each structured data type in an interface should be decomposed into its base types and one assumption for the structure recorded. The base type of the interface should then be examined and additional assumptions recorded.

Here are two examples to illustrate the assumption counting strategy. An array of integers utilized as part of an interface implies an assumption about the interface structure. This assumption should be counted towards that of the module's interface. The minimal entity for this structure is an integer which implies a maximum of two more assumptions (one for the type integer, one for the value the integer may hold) may be made concerning this interface. Thus, a total of three assumptions may be recorded for the array of integers in the module's interface. Consider an array of students where a student is a record consisting of an ID number and a grade. Assumptions may be recorded for the array structure, the record structure, and the ID number and grade for a maximum of six assumptions.

The following algorithm for computing design stability metrics is adapted from Yau and Collofello [YaCo85].

Algorithm for Computing Design Stability

Input: Program design documentation.

Output: $DLRE_x$ and DS_x, $\forall$ module x; and PDS.

Method:

step 1. From the program design documentation, analyze the module invocation hierarchy for the program and $\forall$ module x and identify the following sets:

$J_x = \{$ modules which invoke $x\}$

$K_x = \{$ modules invoked by $x\}$

$R_{xy} = \{$ passed parameters returned from x to module y, where $y \in J_x\}$

$S_{xy} = \{$ parameters passed from x to module y, where $y \in K_x\}$

step 2. From the program design documentation, analyze the program's global data which is defined to consist of global variables and shared files, and $\forall$ module x, identify the following sets:

$GR_x = \{$ global data referenced in $x\}$

$GD_x = \{$ global data defined in $x\}$

from these sets, $\forall$ global data item i, identify the set:

$G_i = \{x \mid i \in (GR_x \bigcup GD_x)\}$

NOTE: Calculation of the set G is undecidable for languages having pointer variables. In such a case, the set G_i is calculated as the worst case, i.e., it includes all global items which may be accessed via the pointers.

`step 3.` $\forall$ set R_{xy} and each parameter $i \in R_{xy}$, find the number of assumptions made by module y about i utilizing the following pseudocode algorithm. (a) If parameter i is a structured data element, then decompose i into its base types and increment the assumption count by 1, else consider i to be a minimal entity. (b) While more base elements can be decomposed, select a base element which is not a minimal entity and decompose it into its base elements and increment the assumption count by 1. (c) For each minimal entity comprising i, if module y makes assumptions about the values which the minimal entity may assume, then increment the assumption count by 2, else increment the assumption count by 1. Set TP_{xy} = the total number of *assumptions* made by y about the parameters in R_{xy}.

`step 4.` $\forall$ set S_{xy} and each parameter $i \in S_{xy}$, find the number of assumptions made by module y about i utilizing the pseudocode algorithm in `step 3`. Set TQ_{xy} = the total number of *assumptions* made by y about the parameters in S_{xy}.

`step 5.` $\forall$ module x and every global data item $i \in GD_x$, find the number of assumptions made about i by other modules in the program. This requires utilization of the set G_i and application of the pseudocode algorithm in `step 3` for each global data item i and every module $y \in (G_i - \{x\})$. Set TG_x = the total number of *assumptions* made by other modules about the global data items in GD_x.

`step 6.` Compute *design logical ripple effect*

$$DLRE_x = TG_x + \sum_{y \in J_x} TP_{xy} + \sum_{y \in K_x} TQ_{xy}, \forall \text{ module } x.$$

`step 7.` Calculate *design stability*

$$DS_x = \frac{1}{1+DLRE_x}, \forall \text{ module } x.$$

step 8. Compute *program design stability*

$PDS = \frac{1}{1+\sum_x DLRE_x}$, where x is a module in the program.

3.2.2 Logical Stability

The *stability* of a program is defined by Yau and Collofello as the quality attribute which indicates the resistance of a program to the potential ripple effect which a program would have when it is modified during software maintenance. This measure is dependent on the stability of the *modules* of the program. The *logical stability* of a module is a measure of the resistance to the impact of a modification of the module on other modules in the program in terms of logical considerations (as opposed to performance considerations) [YaCo80].

The intent of Yau and Collofello is not to use stability measures as indicators of program maintenance, but as significant factors contributing to program maintainability. They propose utilizing this measure in conjunction with other attributes affecting program maintainability. As an illustrative example, they mention that: "... a single program of 20,000 statements will possess an excellent program stability since there cannot be any ripple effect among modules; however, the maintainability of the program will probably be quite poor."

Some important points and definitions are discussed below.

Basis of Analysis. The computation of the logical stability of a module is based on a primitive subset of the maintenance activity for which the impact of the modifications can be readily determined: *a change to a single variable definition of a module.* This choice of a primitive subset of the maintenance activity is justified because, regardless of the complexity of the maintenance activity, it basically consists of modifications to variables in modules.

Aspects of Logical Ripple Effect. There are two aspects of the logical ripple effect which are to be examined. One aspect concerns *intramodule change propagation.* This involves the flow of program changes within the module as a consequence of the modification. The other aspect concerns *intermodule change propagation.* This involves the flow of program changes across module boundaries as a consequence of the modification.

definitions

module interface variables — consist of the module's global variables, its output parameters, and its variables utilized as input parameters to called modules.

unique interface variable — each utilization of a variable as an input parameter to a called module.

worst case logical ripple effect analysis — calculate the set X_{kj} by first identifying all the modules for which j is an input parameter or global variable. Then, for each of these modules in X_{kj}, the *intramodule* change propagation emanating from j is traced to the interface variables within the module. *Intermodule* change propagation is then utilized to identify other modules affected and these are added to X_{kj}. This continues until the ripple effect terminates or no new module can be added to X_{kj}. (See steps 3 and 4 of the algorithm that follows these definitions for computing logical stability measures).

assumptions — A significant refinement to the worst case change propagation can result by utilizing the approach of examining whether or not a module makes any assumptions about the values of its interface variables. These assumptions can be expressed as *program assertions*. If it does not make any assumptions about the values of the interface variables, the modules cannot be affected by intermodule change propagation. However, if it does make an assumption about the value of an interface variable, the worst case is automatically in effect and the module is placed in the change propagation resulting from affecting the interface variable if the interface variable is also in the change propagation as a consequence of some modification.

The following algorithm for computing logical stability metrics is adapted from Yau and Collofello [YaCo80].

Algorithm for Computing Logical Stability Measures

Input: Program.

Output: LRE_x and LS_x, $\forall$ module x; LREP; and LSP.

Method:

step 1. $\forall$ module x, set $V_x = \{$all variable definitions in module $x\}$. Each occurrence of a variable in a variable definition is uniquely identified in V_x. Thus, if the same variable is defined twice within a module, the set V_x contains a unique entry for each definition. The set V_x is created by scanning the source code of module x and adding to V_x all variables which satisfy any of the following criteria. (a) the variable is defined in an assignment statement; (b) the variable is assigned a value which is read as input; (c) the variable is an input parameter to module x; (d) the variable is an output parameter from a called module; or (e) the variable is a global variable.

step 2. $\forall$ module x, set T_x = {all interface variables in module x}. The set T_x is created by scanning the source code of module x and adding to T_x all variables which satisfy any of the following criteria. (a) the variable is a global variable; (b) the variable is an input parameter to a called module. Each utilization of a variable as an input parameter to a called module is regarded as a unique interface variable. Thus, if variable i is utilized as an input parameter in two module invocations, each occurrence of i is regarded as a unique interface variable; or (c) the variable is an output parameter of module x.

step 3. $\forall$ module x and each variable definition $i \in x$, set Z_{xi} = {interface variables in T_x which are affected by a modification to variable definition i of module x by intramodule change propagation}.

step 4. $\forall$ module x and each interface variable $j \in x$, set X_{xj} = {modules in intermodule change propagation as a consequence of affecting interface variable j of module x}.

step 5. $\forall$ module x and each variable definition $i \in x$, compute the set W_{xi} consisting of the set of modules involved in intermodule change propagation as a consequence of modifying variable definition i of module x. Set $W_{xi} = \bigcup_{j \in Z_{xi}} X_{xj}$.

step 6. $\forall$ module x and each variable definition $i \in x$, compute the *logical complexity of modification* $LCM_{xi} = \sum_{t \in W_{xi}} C_t$, where C_t is McCabe's cyclomatic complexity of module t.

step 7. $\forall$ module x and each variable definition $i \in x$, calculate the probability that a particular variable definition i of module x will be selected for modification $P(xi) = \frac{1}{|V_x|}$, $\forall$ variable definition i of module x.

`step 8.` Compute *potential logical ripple effect of a module*

$LRE_x = \sum_{i \in V_x}[P(xi) * LCM_{xi}]$

and *logical stability of a module*

$LS_x = \frac{1}{LRE_x}, \quad \forall$ module x.

`step 9.` Compute *potential logical ripple effect of a primitive modification to the program* $LREP = \sum_{x=1}^{n}[P(x) * LRE_x]$ where $P(x) = \frac{1}{n}$ and n = number of modules in the program.

`step 10.` Compute *logical stability of the program* $LSP = \frac{1}{LREP}$

The following algorithm for computing logical ripple effect may be modified to compute intramodule and intermodule change propagation for the logical stability algorithm. This is adapted from Yau et al. [YaCo78].

STAGE 1 *Lexical Analysis*

I. Compute precedence order for each module defined from the program's invocation graph.

II. $\forall$ module i, scan the module's code and produce a control flow graph based on program blocks. (A *program block* is a maximal set of ordered statements such that it is always executed from the first statement to the last statement and that all the statements are executed if one of them is executed.)

III. Characterize each program block v_i in terms of its *source capable set* C_i, its *potential propagator set* P_i, and a flow mapping $C_i \leftarrow f(P_i)$. C_i is the set of definitions in block v_i which cause potential error to exist within and flow from

v_i. P_i is the set of all usages in v_i which can cause elements in the source capable set to flow from v_i.

Example: for block v_i $X2 = SQRT(-DISC)$, $XR1 = X1$, $XR2 = X1$, and $XI = X2$

$$C_i = \{X2, XR1, XR2, XI\}$$

$$P_i = \{DISC, X1\}$$

$$\{X2, XI\} \leftarrow f(DISC)$$

$$\{XR1, XR2\} \leftarrow f(X1)$$

STAGE 2 *Computing ripple effect*

A set of modules and their primary error sources involved in the initial maintenance task should be supplied.

I. *Intramodule Change Propagation.* This algorithm operates on each module characterization to trace error sources from their points of definition to their exit points. For each module, M_j, initially involved in the modification, trace the intramodule flow of potential errors from the primary error sources through various program blocks. When the flow of error sources stabilizes, apply a block identification criterion to determine which blocks within the module must be examined to insure that they are not inconsistent with the initial change. After block identification is complete, a propagation criterion is applied to module M_j to other modules which M_j invokes, and to modules which invoke M_j.

II. *Intermodule Change Propagation.* Error flow across module boundaries constitutes intermodule error flow. For each module affected by intermodule error flow, the algorithm traces intramodule error flow in the same manner as for

M_j to determine the net effect that the propagated error sources have on their respective modules. The algorithm executes in this manner until intermodule error flow stabilizes.

III. *Ripple Effect.* At this point, the set of modules in the program which are affected by the intermodule flow of error sources created by the primary error sources involved in the maintenance task has been determined. Complete the algorithm by applying a ripple effect criterion to each module affected by intermodule error flow to determine if the module requires additional maintenance activity to insure that the module is not inconsistent with the initial change.

CHAPTER IV

EXPERIMENTATION FRAMEWORK

Basili et al.'s classification scheme for experimentation in Software Engineering [BaSH86] will be employed to present the pre-experimental design used in this study. Each of the four categories: definition, planning, operation, and interpretation, corresponds to a phase of the experiment.

4.1 Definition

The motivation for this prototype study is to understand, assess, and learn more about the product of the literate programming process: WEB programs (object). The purpose is to capture, quantify, and characterize the attributes of a WEB program which contribute to the effort in (a) understanding it, or (b) explaining it to someone else. The major questions are: what needs to be explained and how important are the relative weights of the features particular to the WEB environment? A subsidiary goal is to motivate and promote the literate programming paradigm.

This pre-experimental design takes into account the perspectives of a WEB program developer, modifier, maintainer, user, or researcher — anyone who may have the need to read and understand what the program is all about. The domain of the study consists of several complete WEB programs that have been published by "experts" of literate progamming (i.e., Knuth and Sewell [Knut84, Sewe89]) or developed by a "novice" (i.e., the author). The scope can be classified as multi-project variation [BaSH86].

4.2 Planning

The design of the experiment is a pre-test multi-project variation involving WEB programs which have been published in the literature or developed for testing purposes. The criteria used to evaluate these programs will be objective measurements of size, complexity, and stability metrics. They include lines of code, lines of documentation, several of Halstead's Software Science measures, McCabe's cyclomatic complexity, McCabe and Butler's design complexity, and Yau and Collofello's design stability measure. An additional criteria is the counts of commands specific to the WEB environment.

The metrics calculations are used to study the relationships between the commands specific to the WEB environment and size, complexity, and stability metrics. Preliminary observations based on these relationships will be made in this pilot investigation to identify the attributes of WEB programs which are of particular interest. These should be investigated further in future experiments on WEB programs and literate programming environments.

The data collection process includes the development of a WEB program, WEBmeter, to automatically gather and calculate the size and source code metrics data, as well as the command counts specific to the WEB environment. The remaining design complexity and stability metrics are hand-calculated. The measurement of data taken will either belong to the ordinal scale or the interval scale [CoDS86] depending on the particular metric in question.

4.3 Operation

The operation phase is the third phase of the experimental study. This phase consists of three parts: preparation, execution, and analysis.

4.3.1 Preparation

The preparation for the operation of the experiment included developing a literate program to calculate some of the target metrics and designing algorithms to be used in hand-calculating the remaining ones. The ideal situation would have been to completely automate the entire data collection process. However, due to the difficulty in deriving structure metrics from WEB programs (which are basically free-form in regard to traditional Pascal syntax) and time constraints, full automation was relegated to future work.

A literate program, WEBmeter, was developed on a Sun 3/60 workstation running SunOS release 4.0.3. Knuth's WEB System consisting of the document formatting language TEX (C version 2.93) and the Pascal programming language (Sun Pascal) was chosen because most of the literate programs published in the literature were developed on this system [BeKn86, BeKM86, Knu86a, Knu86b, Knut84, Sewe89]. The preprocessors TANGLE (C version 2.8) and WEAVE (C version 2.9) are also part of this WEB system of structured documentation.

WEBmeter (see Appendix B) expects as input a syntactically correct WEB progam — that is, a program which can be WEAVEd, TEXed, TANGLEd, and compiled with no errors. (It would be desirable to have a CHange file along with the WEB program as input.) The output produced is written to a user-defined file, and includes a list of the operands and operators, with their respective frequencies; η_1, η_2, N_1, and N_2 and other Software Science measures that can be derived from them: length, volume, effort, and time; McCabe's cyclomatic complexity number; as well as the various commands specific to the WEB environment and their frequencies; and finally, size metrics including the number of lines of limbo, lines of documentation, lines of code, and number of macro and format definitions in the WEB program.

A sample input WEB program and its generated output appears in Appendix C. The listings include the WEAVEd, TANGLEd, and WEB source code; a sample execution of the program; and the output generated by WEBmeter.

The implementation of WEBmeter basically consists of a one-pass lexical analyzer which counts most of the Pascal operators and WEB-specific commands and calls the parser to determine if a token is an operand or an operator and whether it affects the cyclomatic complexity number. The Software Science measures are calculated from the four basic counts η_1, η_2, N_1, and N_2. The counting of operands, operators, and the cyclomatic number in WEBmeter is based on Conte et al.'s *Pascal Counting Strategy* [CoDS86]. Design decisions regarding the counting of WEB-related entities are listed in the WEB *Counting Strategy* of WEBmeter (see Appendix B).

Algorithms for computing design complexity and stability metrics were developed as a guide for the hand-calculation process. They may also be used as program design documentation for automating the process.

McCabe and Butler's design and integration complexity measures (see Section 3.1.1) are calculated from a structure chart of the program and the flowgraphs of each module. Applying several reduction rules to a flowgraph will produce the primary control structure for calling subordinate modules. The cyclomatic complexity of the reduced flowgraphs are used to calculate the design complexity, which in turn is used to find the integration complexity.

Yau and Collofello's design stability measures (see Section 3.2.1) are calculated from a program's design documentation. Parameters and global data used in each module are analyzed to determine the total number of assumptions made by the module and its design stability value. The design stability of the program is calculated using the design stability measures of the subordinate modules.

4.3.2 Execution

Three of the six input `WEB` programs used to collect the data (*primes.web*, *knights.web*, and *queens.web*) were taken from the literature. *Primes.web*, published by Knuth [Knut84], was available on-line among the files of the TEX distribution tape already on the Sun. It is a program to print the first 1000 prime numbers. The next two programs, *knights.web* and *queens.web* were published by Sewell [Sewe89]. Because they were not available on-line, they were manually entered. *Knights.web* was copied from the published `WEB` source code and *queens.web* was translated into `WEB` from the published `WEAVE`d listing. Both are classic Computer Science problems. The *Knight's Tour* involves a knight, which can only move according to the rules of chess, trying to move to every square of a chessboard once and only once. The *Eight Queens* problem consists of placing eight queens on a full-size chessboard such that no queen can "check" another queen (each queen must be placed so that it is not on the same row, column, or diagonal as any other queen).

The remaining three input `WEB` programs were developed by the author. *Sample.web* is a simple program which calculates the maximum, minimum, and mean value of an array of real numbers. *Reg.web* is a program which solves a set of regular expression equations in standard form to give the minimal fixed-point solution. The final input program is `WEB`meter itself.

`WEB`meter was `TANGLE`d and compiled so the program could be executed multiple times without recompilation. Two macros **stat** and **tats** are provided which, if not commented out using the WEB meta-comment commands (@{ and @}), will write output to the terminal in a summary format (in addition to the regular output file). The data collection process on the Sun consisted of using the Unix **script** utility to collect the data, which crossed the terminal screen each time WEBmeter was executed with a different input program, into a file called **typescript**. The WEBmeter-generated output for the six input programs is provided in the next section.

Design complexity and stability measures were hand-calculated for the TANGLEd versions of all input programs except WEBmeter itself. Intermediate and final calculations for McCabe and Butler's design complexity (S_0) and integration complexity (S_1), and Yau and Collofello's design stability (PDS) appear in Appendix D. The next section includes a summary of the hand-calculated metrics.

4.3.3 Analysis

According to Basili et al. [BaSH86], the final stage of the Experiment Operation phase is analysis of data. Because of the small number of input programs available for this study, formal data analysis using statistical models and tests was not performed. Analysis of this type is meaningful only when a representative and reasonably-sized sample is used, thus justifying the extrapolation of results to other similar environments. In this prototype investigation of WEB program, "data analysis" will consist of making preliminary observations based on the metrics calculations and the WEB environment specific command counts. These observations may be used to formulate hypotheses to be tested in future experiments.

Table 1 contains the WEB environment specific commands tokenized in the input WEB programs, along with the token names in WEBmeter and their corresponding action.

Table 1. WEB Environment Specific Commands

Command	Token	Action
@@	*tat*	the single character '@'
@␣	*tnew_mod*	new code section ('␣' is a blank)
@*	*tstar_mod*	new starred (major) code section
@d	*tdef*	macro definition
@f	*tformat*	format definition
@p	*tbegin_code*	start Pascal part of an unnamed section
@< @>	*tmod_name*	a code section name
@'	*toctal*	octal constant
@"	*thex*	hexadecimal constant
@$	*tcheck_sum*	string pool check sum
@{ @}	*tbegin_code*	a "meta-comment"
@&	*tjoin*	concatenate two elements with no space
@^ @>	*troman*	roman font index entry
@. @>	*ttypewriter*	typewriter font index entry
@: @>	*tuser_def*	user-controlled font index entry
@t @>	*ttex_string*	pure TEX text
@= @>	*tverbatim*	verbatim text
@\	*tforce_line*	force end-of-line in Pascal text
@!	*tunderline*	underline index entry
@?	*tno_underline*	cancel underline in index
@,	*tthin_space*	insert a thin space in TEX file
@/	*tline_break*	force a line break in TEX file
@|	*t_opt_line_break*	optional line break in TEX file
@#	*t_big_line_break*	force a line break with extra vertical space in TEX file
@+	*t_no_line_break*	cancel a pending line break in TEX file
@;	*tpseudo_semi*	invisible semicolon in TEX file
@x	*tx*	start of a change section (change files only)
@y	*ty*	start of replacement text (change files only)
@z	*tz*	end of a change section (change files only)

Table 2. WEB Environment Features

<table>
<tr><th>Feature</th><th>Token</th><th>Description</th></tr>
<tr><td>| |</td><td>tassumpt</td><td>embedded Pascal code in TEX code</td></tr>
<tr><td>' '</td><td>tstring</td><td>Pascal text string</td></tr>
<tr><td>" "</td><td>tpreproc</td><td>preprocessed string</td></tr>
<tr><td>{ }</td><td>tcomment</td><td>Pascal in-line comment</td></tr>
<tr><td>#</td><td>targument</td><td>macro argument</td></tr>
<tr><td>==</td><td>tdbl_eql</td><td>defining simple or parametric macro</td></tr>
<tr><td>=</td><td>tone_eql</td><td>defining numeric macro</td></tr>
<tr><td>@< @>=</td><td>tis</td><td>defining Pascal part of a code section</td></tr>
<tr><td>@< @>;</td><td>tcall</td><td>calling a code section</td></tr>
</table>

To count some of the additional features of WEB programs that are not WEB commands (a command starts with a '@'), some additional tokens were counted. These are listed in Table 2, along with a short description of each.

Table 3 contains the frequencies of the WEB environment specific commands used in the six input programs.

Fourteen of the twenty-nine commands were not used by any of the three WEB program developers. These include (by token name): *toctal, thex, tcheck_sum, tjoin, ttex_string, tverbatim, tforce_line, tno_underline, tthin_space, tbig_line_break, tno_line_break, tx, ty,* and *tz.* In addition, four WEB commands: *tformat, tbegin_comment* (a meta-comment), *tuser_def*, and *topt_line_break* were used in only one program each. This may be due to the overall simplicity of the applications shared among the input programs (i.e., they did not require special commands such as *toctal, thex, tcheck_sum, tjoin*, etc.).

Upon examining the list of unused text formatting commands, it is reasonable to believe that these will always be used infrequently. In fact, "novice" programmers may never use them, while "experts" may only use them in specific situations. (I

Table 3. WEB Environment Specific Command Counts

WEB Command	Programs					
	sample	queens	primes	knights	reg	WEBmeter
@@	2	2	0	2	0	3
@␣	9	20	21	24	27	84
@*	5	4	6	4	10	12
@d	0	12	5	19	0	20
@f	0	0	0	0	0	6
@p	1	1	1	1	1	1
@< @>	21	22	37	35	64	189
@{ @}	0	0	0	0	0	12
@^ @>	0	8	9	10	0	3
@. @>	0	1	1	2	0	13
@: @>	0	0	0	0	0	1
@!	10	8	17	18	30	127
@/	11	2	1	3	0	15
@|	0	0	0	20	0	0
@;	3	1	2	3	10	19

Table 4. WEB Environment Feature Counts

WEB Feature	Programs					
	sample	queens	primes	knights	reg	WEBmeter
| |	11	33	105	70	76	191
‘ ’	5	1	2	1	16	398
{ }	21	3	25	9	18	95
#	0	4	6	18	0	14
==	0	10	5	16	0	19
=	0	2	0	3	0	7
@< @>=	11	13	23	18	34	84
@< @>;	10	9	14	17	30	105

would venture to guess that if Knuth's WEB programs TeX, *Metafont*, TANGLE, and WEAVE [Knu86a, Knu86b, Knut83, Sewe89] were used as input programs, all of the text formatting commands would have been used at least once.)

The WEB command *tbegin_code* (@p) was used by each program once. Every WEB program is required to use @p at least once, because TANGLE uses this unnamed section to start "building" its Pascal program. Multiple unnamed code sections work in the same manner as multiple named sections: they are appended in the order they appear in the WEB source code.

Table 4 contains the counts of the additional WEB environment features tokenized, except for *tpreproc.*

Preprocessed strings were not used in any of the six input programs. What follows is a short explanation of why each feature was selected for counting. The *tassumpt* count refers to the number of times Pascal code appears in the TeX part of the code sections of a WEB program. This may be used to measure the number of "assumptions" of the program, because an occurrence of the token signals a direct explanation of the Pascal code or variable referenced. The count of the token *tstring* simply measures the number of Pascal text strings used. `WEB`meter's *tstring* count is relatively high because of the large amount of output generated, both normal and debug. The *tcomment* count represents the number of Pascal in-line comments in the `WEB` program. Because the `WEB` System provides for separate commentary in each code section, it would be logical to hypothesize that `WEB` programs contain fewer in-line comments than traditional Pascal programs. The next three tokens are directly related to macros. The count of *targument* gives an estimate of the number of arguments used in the parametric macros of the program. The *tdbl_eql* count is the total number of simple and parametric macros defined, while *tone_eql* is the number of numeric macros. These two should add up to the count of *tdef* in Table 3. The final two features break down the *tmod_name* (`@< @>`) count in Table 3. Token *tis* (`@< @>=`) is used when defining the replacement Pascal part of a code section. *Tcall* (`@< @>;` or `@< @>@;`) can be described as a "call" to a code section. It signals `TANGLE` to replace the code section name with the section's corresponding Pascal code. These are analogous to defining a procedure vs. calling a procedure in traditional Pascal programs.

The size metrics generated by `WEB`meter for the six input programs appear in Table 5. These measures include the number of identifier tokens recognized (`TIDENT`), the number of number tokens recognized (`TNUM`), the total number of numbered code sections in the `WEB` program (`CS`), the number of procedures defined (`PROC`), the number of functions defined (`FUNCT`), the number of lines of limbo text (`LOL`), the number of lines of documentation in the code sections of the program (`LOD`), the average number of lines of documentation per code section (`LOD/CS`), the number of lines used to define macros and format statements (`LOM`), the number of lines of code used in the code sections of the program (`LOC`), and finally the average number of lines of code per

Table 5. Size Metrics

Size Metric	WEB Programs					
	sample	queens	primes	knights	reg	WEBmeter
TIDENT	71	89	117	140	562	2252
TNUM	19	24	30	59	98	307
CS	14	24	27	28	37	96
PROC	1	1	0	1	4	6
FUNCT	1	0	0	0	4	9
LOL	9	13	16	14	14	10
LOD	49	133	233	263	171	506
LOD/CS	3.50	5.54	8.63	9.39	4.62	5.27
LOM	0	12	5	19	0	26
LOC	78	66	91	110	365	1312
LOC/CS	7.09	5.08	3.96	6.11	10.74	15.62

code section (LOC/CS). Note that the two averages are computed differently. LOC/CS only counts code sections which contain a Pascal part. This number is reflected in the count of token *tis*. Also, the LOC count excludes blank lines, but includes in-line comments. In addition, CS is the sum of the *tnew_mod* and *tstar_mod* counts from Table 3.

Notice that the WEB programs listed in Table 5 are ordered by increasing size (see counts of TIDENT, TNUM, and CS). The only aberrations are the LOL count for *knights*, the LOD and LOD/CS measures for *reg*, the LOD/CS for WEBmeter, the LOC and LOC/CS for *sample*, and the LOC/CS count for *primes*.

The LOL count is not very meaningful unless there is a big discrepancy signalling special formatting requirements or many comments. The lower LOD, and LOD/CS values for *reg* and WEBmeter may be attributed to the novice programmer syndromes of underdocumenting and inefficient coding. Documenting a WEB program is difficult when one is unaccustomed to explaining programs in detail. Traditionally accepted amounts of external and internal documentation of Pascal programs does not prepare one adequately for the extensive detailed explanation which may be placed in a WEB program. The LOC and LOC/CS counts were lower for *queens* than for *sample*. Hardly anyone would claim that *sample* is a more difficult program than *queens*. The apparent discrepancy may be due to the fact that the LOL, LOD, LOM, and LOC counts are highly dependent on programming style since they are based on physical lines in the WEB source code. For example, one programmer may type in his/her Pascal code one statement per line while another may disregard this traditional method because WEAVE will format it according to its own conventions. The final discrepancy, the relatively small value of LOC/CS of *primes* may be attributed to the fact that Knuth was using this program to explain features of WEB and the literate programming paradigm, resulting in more documentation and code sections than normal.

Some general observations can be made about the size metrics that were calulated. For example, the number of procedures and functions used were very few. The WEB programmers opted to use macros and code sections over the traditional procedures and functions. Sewell [Sewe89] suggests that procedures only be used when necessary (i.e., parameters must be passed or recursion must be used). He also provides guidelines for using macros vs. code sections. Inexperienced WEB programmers should exercise caution because it is very easy to lose sight of these guidelines. This may lead to an unclean design (i.e., excessive use of global variables, code sections, etc.). Sewell suggests that approximately a dozen lines of code should be a reasonable size for a code section (he does not give an estimate for LOD). None of the programs examined reached this goal except for WEBmeter (which surpassed it). This may indicate the use of too many code sections. The ratios of LOD/CS to LOC/CS vary from approximately 1:3 (WEBmeter) to 2:1 (*primes*).

Table 6. Source Code Complexity Metrics

Code Metric	WEB Programs					
	sample	queens	primes	knights	reg	WEBmeter
VG	8	11	12	13	62	235
η_1	33	37	32	51	48	86
η_2	25	21	32	33	67	498
N_1	156	160	201	268	911	3973
N_2	90	74	135	148	623	2625
LENGTH	246.00	234.00	336.00	416.00	1534.00	6598.00
VOLUME	1441.06	1370.77	2016.00	2659.20	10500.98	60634.46
EFFORT	85599.16	89360.99	136080.00	304116.24	2343442.97	13743202.71
TIME (s)	4755.51	4964.50	7560.00	16895.35	130191.28	763511.26
TIME (m)	79.26	82.74	126.00	281.59	2169.85	12725.19
TIME (h)	1.32	1.38	2.10	4.69	36.16	212.09

The source code complexity metrics generated by WEBmeter are in Table 6. These include cyclomatic complexity (VG), Software Science basic counts η_1, η_2, N_1, and N_2, and the calculated measures LENGTH, VOLUME, EFFORT, and TIME. The majority of the complexity measures increase with size. Exceptions are the η_1 counts of *queens* and *primes*; and the N_2, LENGTH, and VOLUME measures of *sample* and *queens*. These may be attributed to the difference in the number of procedures, functions, and macros declared and called.

Table 7 gives the hand-calculated counts of program design stability (PDS), program design complexity (S_0), and integration complexity (S_1) that were calculated from the TANGLEd versions of the input programs.

A module (defined as a procedure or function) is the basic entity for each of these metrics. As mentioned earlier, the input programs utilized very few modules in their implementation. Thus these may not totally reflect the intentions of the metrics.

Table 7. Design Complexity and Stability Metrics

Design Metric	TANGLEd WEB Programs				
	`sample`	`queens`	`primes`	`knights`	`reg`
PDS	$\frac{1}{17}$	$\frac{1}{21}$	1	$\frac{1}{16}$	$\frac{1}{96}$
S_0	3	2	1	2	23
S_1	1	1	1	1	15

However, they may illustrate the need for improved guidelines for WEB environment specific command and attribute usage.

The PDS counts for *sample.p*, *queens.p*, and *knights.p* are fairly straightforward. Each of these programs were efficient in their use of parameters. *Primes.p* has a PDS count of one because no procedures or functions were used. *Reg.p* had the worst program design stability. This is mainly attributable to the larger number of procedures and functions used, and the parameters which included arrays of records of pointers. Usage of these types of parameter tends to increase assumption counts.

For *sample.p*, *queens.p*, and *knights.p*, $S_0 = n$, where n is the total number of procedures and functions. This means that no subordinate modules were called in conditional statements. Thus $S_1 = 1$ for these three input programs. For *primes.p* since $n = 1$, $S_0 = S_1 = 1$. For the program *reg.p*, the values of S_0 and S_1 reflect the complicated structure chart of the program.

4.4 Interpretation

This is a pre-test, pre-experimental study. One way to validate the observations, would be to obtain a subjective rating of the complexity and stability of the programs by WEB program "experts" and compare the results with the objective data. Also, ideally, it would be necessary to repeat the experiment with a larger number of test programs and a larger group of people.

CHAPTER V

SUMMARY, CONCLUSIONS, AND FUTURE WORK

The literate programming paradigm is a novel approach to developing software. Its fundamental concept equates a computer program to a piece of literature. A person should be able to read a WEB program as if it were a book. Various commands and features unavailable in traditional programming systems are provided by the WEB System of Structured Documentation to enable a developer to easily and automatically create "works of literature." The goal is to help the human reader comprehend the programs quickly and thoroughly.

A prototype study of WEB program was conducted to investigate what features particular to the WEB environment contribute most to the process of explaining or understanding it. A WEB program, WEBmeter, was developed to isolate and count all of the features and commands of interest. In addition, size and code complexity metrics were generated by WEBmeter. Three additional design complexity and stability measures were hand-calculated from the TANGLEd versions of the input WEB programs.

The metrics calculations and the counts of the WEB environment specific commands and features for the set of six input programs were analyzed to make preliminary observations which may be used as a basis for future experiments on WEB programs and literate programming environments. While using functions and procedures are the only way to modularize traditionally developed Pascal programs, WEB programs have additional macro and code sections definition and calling capabilities.

Measuring the relationships between these three mechanisms and how they decrease the perceived complexity of a WEB program for a potential reader seems to be of central importance. The size, code complexity, and design metrics calculated need to incorporate these relationships in order to truly measure the complexity and stability of WEB programs.

From my own experiences in developing the WEBmeter program, I found that the literate programming paradigm did simplify the process of developing and understanding large programs. At first, it was difficult to document and develop the code simultaneously; this became easier as I gained more experience. The ability to explain design decisions and call a code section before actually defining it were definite advantages. The best part was being able to rearrange the code sections to reflect the improved design very easily and without fearing that the program would never run again.

On the down side, I found that writing the rather small program named *sample* was quite tedious due to the overhead of using TeX and WEB commands. When developing larger and complex applications, the overhead is offset by the benefits gained. Also, I tended to overuse code sections (i.e., I sometimes used code sections where procedures or functions would have been more appropriate). This was due mostly to lack of experience in "thinking in WEB." Later I became spoiled by the "extras" (including a nicely typeset listing) that the literate programming paradigm offers. It will be difficult to face a regular Pascal program again, let alone write one.

What follows is a list of possible extensions to this thesis and other areas of related future work.

- To automate the metrics collection process completely.

- To add metrics calculations capabilities to WEBmeter for Fan-in/Fan-out (see Section 3.1.2) and Logical Stability (see Section 3.2.2).

- To form hypotheses based on the observations and to test these hypotheses in large-scale experiments across literate programming environments that utilize various languages and text formatters.

- To define metrics specially designed for literate programming environments to measure the degree of "literateness" of a program. Such metrics can be generalizations of the existing design/code complexity measurements.

- To collect and classify WEB programs into a repository to be used as data in large-scale experiments.

- To design and carry out experiments to examine how WEB programs promote reusability and ease of maintenance.

REFERENCES

[AvOp90] A. Avenarius and S. Oppermann, "FWEB: A Literate Programming System for Fortran 8x," *ACM SIGPLAN Notices*, vol. 25, no. 1, pp. 52–58, January 1990.

[BaSH86] V.R. Basili, R.W. Selby, and D.H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 7, pp. 733–743, July 1986.

[BaZw80] A.L. Baker and S.H. Zweben, "A Comparison of Measures of Control Flow Complexity," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 506–512, November 1980.

[BeEv79] L.A. Belady and C.J. Evangelisti, "System Partitioning and Its Measure," IBM Research Report RC7560, 1979.

[BeGr87] J. Bentley and D. Gries, "Programming Pearls — Abstract Data Types," *Communications of the ACM*, vol. 30, no. 4, pp. 284–289, April 1987.

[BeKn86] J. Bentley and D.E. Knuth, "Programming Pearls — Literate Programming," *Communications of the ACM*, vol. 29, no. 5, pp. 364–369, May 1986.

[BeKM86] J. Bentley, D.E. Knuth, and D. McIlroy, "Programming Pearls — A Literate Program," *Communications of the ACM*, vol. 29, no. 6, pp. 471–483, June 1986.

[Broo75] F.P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering.* Reading, MA: Addison-Wesley, 1975.

[BrCh90] M. Brown and B. Childs, "An Interactive Environment for Literate Programming," *Structured Programming*, vol. 11, pp. 11–25, 1990.

[BrCh89] M. Brown and B. Childs, "An Interactive Tool for Literate Programming," *Third Workshop on Empirical Studies of Programmers*, Austin, April 29-30, 1989.

[Bro88a] M.E. Brown, "An Interactive Environment for Literate Programming," Ph.D. Dissertation, Dept. of Computer Science, Texas A&M University, Technical Report, August 1988.

[Bro88b] M.E. Brown, "The Literate Programming Tool," Computer Science Department, Texas A&M University, Technical Report, August 1988.

[CoDS86] S.D. Conte, H.E. Dunsmore, and V.Y. Shen, *Software Engineering Metrics and Models.* Menlo Park, CA: Benjamin/Cummings, 1986.

[CuSK89] B. Curtis, S.B. Sheppard, E. Kruesi–Bailey, J. Bailey, and D.A. Boehm–Davis, "Experimental Evaluation of Software Documentation Formats," *The Journal of Systems and Software*, vol. 9, pp. 167–207, February 1989.

[Hals77] M.H. Halstead, *Elements of Software Science*. New York, NY: Elsevier North-Holland, 1977.

[Hals79] M.H. Halstead, "Advances in Software Science," in *Advances in Computers, vol. 18*. New York, NY: Academic Press, pp. 119–172, 1979.

[HeKa81] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 5, pp. 510–518, September 1981.

[KaHe81] D. Kafura and S. Henry, "Software Quality Metrics Based on Interconnectivity," *The Journal of Systems and Software*, vol. 2, pp. 121–131, 1981.

[Knu86a] D.E. Knuth, *Computers and Typesetting, Volume B,* TEX: The Program. Reading, MA: Addison-Wesley, 1986.

[Knu86b] D.E. Knuth, *Computers and Typesetting, Volume D, Metafont: The Program.* Reading, MA: Addison-Wesley, 1986.

[Knut84] D.E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, May 1984.

[Knut89] D.E. Knuth, "The Errors of TEX," *Software—Practice and Experience*, vol. 19, no. 7, pp. 607–685, July 1989.

[Knut83] D.E. Knuth, "The WEB System of Structured Documentation," Dept. of Computer Science, Stanford University, Technical Report, 1983.

[Leca85] O. Lecarme, "Literate Programming," *Computing Reviews*, vol. 26, no. 1, p. 75, January 1985.

[Lin89a] C.A. Lins, "A First Look at Literate Programming," *Structured Programming*, vol. 10, no. 1, pp. 60–62, 1989.

[Lin89b] C.A. Lins, "An Introduction to Literate Programming," *Structured Programming*, vol. 10, no. 2, pp. 107–111, 1989.

[McCa76] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, December 1976.

[McBu89] T.J. McCabe and C.W. Butler, "Design Complexity Measurement and Testing," *Communications of the ACM*, vol. 32, no. 12, pp. 1415–1425, December 1989.

[Mitc88] R. Mitchell, "Literate Programming (Knuth)," Ph.D. Dissertation, Council for National Academic Awards, United Kingdom, 1988.

[Papp90] T.L. Pappas, "Literate Programming for Reusability: A Queue Package Example," in *Proceedings of the Eighth Annual Conference on Ada Technology*, Atlanta, pp. 500–514, March 5-8, 1990.

[PaZv83] G. Parikh and N. Zvegintzov, *Tutorial on Software Maintenance.* Silver Spring, MD: Computer Society Press, 1983.

[RaMe88] B. Ramamurthy and A. Melton, "A Synthesis of Software Science Measures and the Cyclomatic Number," *IEEE Transactions on Software Engineering*, vol. 14, no. 8, pp. 1116–1121, August 1988.

[Ram89a] N. Ramsey, "A Spider User's Guide," Dept. of Computer Science, Princeton University, Technical Report, August 1989.

[Ram89b] N. Ramsey, "The Spidery WEB System of Structured Documentation," Dept. of Computer Science, Princeton University, Technical Report, August 1989.

[ReSk89] T. Reenskaug and A.L. Skaar, "An Environment for Literate Smalltalk Programming," in *OOPSLA '89 Proceedings*, pp. 337–345, October 1989.

[Sewe89] W. Sewell, *Weaving a Program: Literate Programming in* WEB. New York, NY: Van Nostrand Reinhold, 1989.

[Thim86] H. Thimbleby, "Experiences of 'Literate Programming' using cweb (a variant of Knuth's WEB)," *The Computer Journal*, vol. 29, no. 1, pp. 201–211, March 1986.

[VWHC88] C.J. Van Wyk, E. Hamilton, and D. Colner, "Literate Programming — Expanding Generalized Regular Expressions," *Communications of the ACM*, vol. 31, no. 12, pp. 1376–1385, December 1988.

[VWHG87] C.J. Van Wyk, D.R. Hanson, and J. Gilbert, "Literate Programming — Printing Common Words," *Communications of the ACM*, vol. 30, no. 7, pp. 594–599, July 1987.

[VWJW87] C.J. Van Wyk, M. Jackson, and D.W. Wall, "Literate Programming — Processing Transactions," *Communications of the ACM*, vol. 30, no. 12, pp. 1000–1010, December 1987.

[VWLT89] C.J. Van Wyk, D.C. Lindsay, and H. Thimbleby, "Literate Programming — A File Difference Program," *Communications of the ACM*, vol. 32, no. 6, pp. 740–755, June 1989.

[VWRa89] C.J. Van Wyk and N. Ramsey, "Literate Programming — Weaving A Language-Independent WEB," *Communications of the ACM*, vol. 32, no. 9, pp. 1051–1055, September 1989.

[VWyk90] C.J. Van Wyk, "Literate Programming: An Assessment," *Communications of the ACM*, vol. 33, no. 3, pp. 361–365, March 1990.

[YaCo78] S.S. Yau, J.S. Collofello, and T.M. MacGregor, "Ripple Effect Analysis of Software Maintenance," in *Proc. of COMPSAC 78*, pp. 60–65, 1978.

[YaCo80] S.S. Yau and J.S. Collofello, "Some Stability Measures for Software Maintenance," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 545–552, November 1980.

[YaCo85] S.S. Yau and J.S. Collofello, "Design Stability Measures for Software Maintenance," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 9, pp. 849–856, September 1985.

APPENDICES

APPENDIX A

ANNOTATED BIBLIOGRAPHY

An Annotated Bibliography of Literate Programming

A. Avenarius and S. Oppermann, "FWEB: A Literate Programming System for FORTRAN 8x," *ACM SIGPLAN Notices*, vol. 25, no. 1, pp. 52–58, January 1990.
This paper is a general discussion of several aspects of literate programming. Topics include the merits of documentation; writing literate FORTRAN programs; FWEB's pretty-printing style; how the literate programming paradigm should be applicable to all non-machine-oriented programming languages; and comments about the future of literate programming. The authors believe that documentation will eventually change from normal linear text to a dynamic hypermedia format. In addition, a short sample FWEB program is provided.

J. Bentley and D. Gries, "Programming Pearls — Abstract Data Types," *Communications of the ACM*, vol. 30, no. 4, pp. 284–289, April 1987.
This paper presents a literate program which is a solution by Gries to the problem of printing the n most common words in a text file (see Bentley, Knuth, and McIlroy below). Gries' criticism of Knuth's implementation is his description of the hash table. In his solution, Gries distinguishes between an abstract data object and the data structure it is implemented by. He describes as his goal the construction of libraries of modules which can be reused as "off-the-shelf" parts.

J. Bentley and D.E. Knuth, "Programming Pearls — Literate Programming," *Communications of the ACM*, vol. 29, no. 5, pp. 364–369, May 1986.
This paper presents an introduction to the literate programming style and Knuth's WEB system. In addition, a small example literate program written by Knuth to output a sorted list of m random integers is included which illustrates some of the features. Bentley credits Knuth for making three fundamental contributions to the area of literate programming: defining and naming the area, creating the WEB system, and providing a body of literate programs (TeX and *Metafont*, to name two).

J. Bentley, D.E. Knuth, and D. McIlroy, "Programming Pearls — A Literate Program," *Communications of the ACM*, vol. 29, no. 6, pp. 471–483, June 1986.
This paper presents a literate program solution to the problem of printing the n most common words in a text file. In his review, McIlroy applauds some features of WEB, but dislikes the solution because of the lack of reuse involved. He demonstrates how the same problem can be solved using utilities in a six line UNIX shell script. Bentley is impressed by Knuth's implementation and description of the new hash trie data structure.

M. Brown and B. Childs, "An Interactive Environment for Literate Programming," *Structured Programming*, vol. 11, pp. 11–25, 1990.
This paper presents an overview of literate programming and the WEB system. A discussion of the current environment that literate programmers use leads to a proposal for a full-scale Literate Programming Environment (LPE). The proposed LPE would automate the literate programming process thereby simplifying the development process. It would include a main control panel, a WEB Editor, a WEB-based Debugger, and a Personal Preference Database. Two typical programming scenarios are presented to illustrate the use of LPE. A prototype of the LPE has been implemented on a Sun.

M. Brown and B. Childs, "An Interactive Tool for Literate Programming," *Third Workshop on Empirical Studies of Programmers*, Austin, April 29-30, 1989.

This paper describes an empirical study involving a prototype literate programming tool developed to investigate whether a hypertext presentation would be useful in a proposed Literate Programming Environment (LPE). A study of a senior-level Computer Science class familiar with the `WEB` System showed a preference for using the prototype tool over the `WEAVEd` (woven?) typeset listing, which was in turn preferred over a standard editor (and `WEB` source code).

M.E. Brown, "An Interactive Environment for Literate Programming," Ph.D. Dissertation, Dept. of Computer Science, Texas A&M University, Technical Report, August 1988.
This dissertation discusses the prototype `WEB` editor developed to investigate whether the development of a Literate Programming Environment would promote the use of and reduce the complexity of using Knuth's `WEB` system. An empirical study of a number of programmers using the `WEB` editor to perform maintenance tasks showed that the `WEB` editor was preferred over the TeX document typesetting system and a regular editor.

M.E. Brown, "The Literate Programming Tool," Computer Science Department, Texas A&M University, Technical Report, August 1988.
This technical report is the `CWEB` source code of the prototype literate programming tool mentioned in the previous three entries (see above). The source code includes three separate programs: Web_Read, Lpt_Server, and Lpt_Client.

P.J. Denning, "Announcing Literate Programming," *Communications of the ACM*, vol. 30, no. 7, p. 593, July 1987.
This is an introduction to the regular column on Literate Programming appearing in *CACM*. Jon Bentley proposed the idea, after receiving favorable reponse to his Programming Pearls columns, which introduced the subject. The literate programming column is published as an experiment to see how much interest is generated and sustained, with a planned evaluation set for mid-1988. The moderator for the column, Christopher J. Van Wyk, is introduced, with a call for interested literate program authors and critics to contact him.

D.E. Knuth, *Computers and Typesetting, Volume B, TeX: The Program.* Reading, MA: Addison-Wesley, 1986.
This 594-page book contains the source code for TeX, developed using the `WEB` System of structured documentation. Also included are a selected Bibliography of TeX and `WEB` references, and a short introduction on how to read the (`WEB`) program.

D.E. Knuth, *Computers and Typesetting, Volume D, Metafont: The Program.* Reading, MA: Addison-Wesley, 1986.
This 560-page book contains the source code for *Metafont*, developed using the `WEB` System of structured documentation. Also included are a selected Bibliography of TeX and `WEB` references, and a short introduction on how to read the (`WEB`) program.

D.E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, May 1984.
This seminal paper is the first published article introducing the concept of literate programming. The philosophy and features of the `WEB` programming language and documentation system are described. Also included is a sample literate program for printing the first 1000 prime numbers. Other topics discussed are the portability

of WEB, advantages of programming in WEB, stylistic and economic issues, and future implications.

D.E. Knuth, "The WEB System of Structured Documentation," Dept. of Computer Science, Stanford University, Technical Report, 1983.
This technical report is a User Manual for WEB. It describes how to write programs in the WEB language using WEB control codes, TEX text, and Pascal code. Appendices include an example excerpt of a WEB program with its WEAVE and TANGLE outputs; the complete WEB source code of WEAVE and TANGLE; and instructions for installing the WEB system.

O. Lecarme, "Literate Programming," *Computing Reviews*, vol. 26, no. 1, p. 75, January 1985.
This is a review of Knuth's introductory paper with the same name. Both positive and negative aspects of the WEB system are discussed along with suggestions for improvements.

C.A. Lins, "A First Look at Literate Programming," *Structured Programming*, vol. 10, no. 1, pp. 60–62, 1989.
This paper is a first in a series of regular columns in the journal of *Structured Programming* on literate programming. The problem of facilitating readability and understandability of programs and their documentation is discussed, with literate programming named a possible solution. The philosophy of literate programming and Knuth's WEB system are introduced. Some issues to be considered and possible research directions are listed.

C.A. Lins, "An Introduction to Literate Programming," *Structured Programming*, vol. 10, no. 2, pp. 107–111, 1989.
This paper begins with a brief description of each element of a literate program section (commentary or description, macros, and code) and the indexing facility. A sample literate program describing an interface to an abstract data type is presented. The author uses Modula-2 and a brute-force method because at the time he had no access to a WEB system. The problems caused by non-automatic generation of literate programs are discussed.

R. Mitchell, "Literate Programming (Knuth)," Ph.D. Dissertation, Council for National Academic Awards, United Kingdom, 1988.
This dissertation extends the idea of literate programming by applying the principle of separation of concerns. An approach to program development based on data abstraction and a formal specification language is taken. More specifically, the programming language Modula-2 and the specification language OBJ are used. [abstract]

T.L. Pappas, "Literate Programming for Reusability: A Queue Package Example," in *Proceedings of the Eighth Annual Conference on Ada Technology*, Atlanta, pp. 500–514, March 5-8, 1990.
This paper begins with a set of guidelines for writing and documenting reusable Ada software. AdaWeb, a literate programming system combining Ada and TEX is described. A sample AdaWeb package, *Bounded Generic Queue Package*, is provided. Features of AdaWeb are explained as they are used in the literate program.

N. Ramsey, "A Spider User's Guide," Dept. of Computer Science, Princeton University, Technical Report, August 1989.

This manual explains how to use the Spider program to generate a WEB system for any programming language. It describes the syntax of the Spider description file used to describe a programming language, giving several examples. It does not say how to use the generated WEB system. [abstract]

N. Ramsey, "The Spidery WEB System of Structured Documentation," Dept. of Computer Science, Princeton University, Technical Report, August 1989.
This manual describes how to write programs in the WEB language using WEB systems generated by Spider. Most of the material is taken verbatim from Donald Knuth's original memo introducing WEB (see Knuth, technical report above). It contains a brief introduction to the idea of literate programming, a short explanation of how to run WEAVE and TANGLE, and a list of all the control sequences that can be used in WEB programs and their effects. [abstract]

T. Reenskaug and A.L. Skaar, "An Environment for Literate Smalltalk Programming," in *OOPSLA '89 Proceedings*, pp. 337–345, October 1989.
The programming environment described in this paper is an adaptation of Knuth's concept of literate programming applied to Smalltalk programs. The environment provides a multi-media document production system including media for Smalltalk class and method definitions. [abstract]

W. Sewell, *Weaving a Program: Literate Programming in* WEB. New York, NY: Van Nostrand Reinhold, 1989.
This book is the first and only book published on literate programming or the WEB System to date. Part I is a User's Guide, which includes an introduction to literate programming, the WEB language, TeX, variant WEB systems, WEB utilities, and a review of some of the current research being done in the area. Part II, titled "Advanced Topics" includes information on porting WEB to different environments and how to tailor WEB to personal preferences. The appendices provide several sample programs (in Pascal, C, and Modula-2) and useful utilities as well as command summaries, and the source code for TANGLE and WEAVE.

H. Thimbleby, "Experiences of 'Literate Programming' using cweb (a variant of Knuth's WEB)," *The Computer Journal*, vol. 29, no. 1, pp. 201–211, March 1986.
This paper presents Thimbleby's version of Knuth's WEB system, called cweb. Cweb uses the programming language C coupled with the text formatting language troff. A description of how cweb works and an excerpt of a cweb program are included. Also discussed are the advantages and disadvantages of literate programming, possibilities of using cweb as a trivial IPSE (Integrated Project Support Environment), ideas for an interactive version of cweb, possible extensions, and implementation problems.

C.J. Van Wyk, "Literate Programming: An Assessment," *Communications of the ACM*, vol. 33, no. 3, pp. 361–365, March 1990.
This column presents a review of the regular column on Literate Progamming appearing in *CACM* since July 1987. Van Wyk lists three aspects common to Knuth's published literate programs: cosmetic details, polish, and verisimilitude (exactly the same input that is used to prepare the program is published). He notes that the four programs appearing in the column achieved the first two aspects, but not the third. The versions used to publish the programs were produced after the code was written. Van Wyk concludes by stating that only programs produced from literate programming systems will be published in future columns; and the column will only

continue if there is interest by literate program developers who use systems they have not designed themselves.

C.J. Van Wyk, E. Hamilton, and D. Colner, "Literate Programming — Expanding Generalized Regular Expressions," *Communications of the ACM*, vol. 31, no. 12, pp. 1376–1385, December 1988.
This paper presents a program by Hamilton that was developed using tools which interleave code and design information. No further information about the specifics of the tools or literate programming style is mentioned. The program is reviewed by Colner, who suggests improvements which would remove the limitations of Hamilton's program.

C.J. Van Wyk, D.R. Hanson, and J. Gilbert, "Literate Programming — Printing Common Words," *Communications of the ACM*, vol. 30, no. 7, pp. 594–599, July 1987.
This paper presents Hanson's solution to the problem of printing the n most common words in a text file (see Bentley, Knuth, and McIlroy; and Bentley and Gries above). The program is written in C and presented using the loom system. Loom is a preprocessor that takes a text file with references to program segments and integrates them with the actual program fragments to produce output, which in turn becomes input to a document formatter (such as TEX). A review of all three solutions is given by Gilbert. His view is that literate programming has different meanings in different circumstances. According to Gilbert, "It is not a matter of artistry or efficiency alone; it is more a question of suitability in context."

C.J. Van Wyk, M. Jackson, and D.W. Wall, "Literate Programming — Processing Transactions," *Communications of the ACM*, vol. 30, no. 12, pp. 1000–1010, December 1987.
This paper presents a program written by Jackson using the JSP design method. The methodology is explained in the commentary portion located at the end of each section. The program is reviewed by Wall. He determines that Jackson's methodology is useful in data processing applications where program structure depends on the structure of the data; but perhaps not quite useful for applications where input/output is not as strictly defined.

C.J. Van Wyk, D.C. Lindsay, and H. Thimbleby, "Literate Programming — A File Difference Program," *Communications of the ACM*, vol. 32, no. 6, pp. 740–755, June 1989.
This paper begins with a traditional C program (which is not claimed to be a literate program by Knuth's definition). Lindsay wishes to demonstrate how a well-written program can be attained without using a WEB system by using standard programming technology only. His work is reviewed by Thimbleby, who believes that the literate programming paradigm provides all the desireable features of a literate program automatically, and for "free". The same results could not be reached by simulating literate programming unless considerably more effort is expended, and with no guarantee of being free of errors.

C.J. Van Wyk and N. Ramsey, "Literate Programming — Weaving a Language-Independent WEB," *Communications of the ACM*, vol. 32, no. 9, pp. 1051–1055, September 1989.

This paper describes how the program Spider enables literate programmers to use a variant of Knuth's **WEB** system by combining TeX with a programming language, X, of their choice (instead of Pascal). The description of the targeted programming language is combined with language-independent master parts of **TANGLE** and **WEAVE** to produce C code for **XTANGLE** and **XWEAVE**. The major differences and benefits of the Spider generated versions of **XTANGLE** and **XWEAVE** compared to Knuth's versions are discussed.

APPENDIX B

WEBmeter

WEBmeter

1. Introduction. WEBmeter was developed by Lisa M. C. Smith, on a Sun 3/60 workstation running SunOS release 4.0.3, TeX C version 2.93, TANGLE C version 2.8, WEAVE C version 2.9, and Sun Pascal.

This program computes various size and complexity metrics as detailed in the text (see `metrics definitions` in Index). This program assumes that the input WEB program is syntactically correct. That is, it can be WEAVEd, TeXed, TANGLEed, and compiled with no errors.

The "banner line" defined here should be changed whenever this program is modified.

define *banner* ≡ `'This␣is␣WEBmeter,␣version␣1.0'`

2. Here is a skeleton of the program:

program *webmeter*(*input*, *output*, *web_file*, *met_file*);
  const ⟨ Global constants of the program 21 ⟩
  type ⟨ Global types of the program 19 ⟩
  var ⟨ Global variables of the program 13 ⟩
    ⟨ Procedure halstead 10 ⟩
    ⟨ Procedure mccabe 14 ⟩
    ⟨ Utilities for input and output 84 ⟩
    ⟨ Utilities for lexical analysis 60 ⟩
    ⟨ Procedures for lexical analysis 25 ⟩
    begin *writeln*; *writeln*(*banner*); *writeln*; ⟨ Calculate complexity metrics 3 ⟩
    ⟨ Output metrics 4 ⟩
    end.

3. ⟨ Calculate complexity metrics 3 ⟩ ≡
  ⟨ Initialize arrays 22 ⟩;
  ⟨ Initialize global variables 64 ⟩
  ⟨ Perform lexical analysis 24 ⟩;
  ⟨ Calculate software science measures 11 ⟩;

This code is used in section 2.

4. ⟨ Output metrics 4 ⟩ ≡
  outcounts;

This code is used in section 2.

5. Some of the code is optional. The material enclosed between the delimiters **debug** and **gubed**, is only included in the TANGLEd source code when the macro definitions are set to an empty statement. The **stat** and **tats** statements work in the same manner. These will be used when collecting summary data. It is highly recommended that if **stat** is true, **debug** should be false (only use one at a time).

define *debug* ≡ `@{` { change this to '*debug* ≡ ' when true }
define *gubed* ≡ `@}` { change this to '*gubed* ≡ ' when true }
define *stat* ≡ { change this to '*stat* ≡ ' when true }
define *tats* ≡ { change this to '*tats* ≡ ' when true }
format *debug* ≡ *begin*
format *gubed* ≡ *end*
format *stat* ≡ *begin*
format *tats* ≡ *end*

6. Here are some macros for common programming constructs.

define *incr*(#) ≡ # ← # + 1 { increase a variable by unity }
define *decr*(#) ≡ # ← # − 1 { decrease a variable by unity }
define *do_nothing* ≡ { empty statement }

7. Sun Pascal **case** statements may include a default case that applies if no matching label if found. To make the program more portable, a macro to rename the reserved word will be used. In addition, a macro to rename the reserved word *external* will be defined to encourage portability.

define *othercases* ≡ *otherwise* { default for cases not explicitly listed }
format *othercases* ≡ *else*
define *extern* ≡ *external* { externally defined procedures and functions }
format *extern* ≡ *forward*

8. The input will come from *web_file*. An enhancement to the program would be to enable an optional CHange file to be input along with the *web_file*.

A CHange file includes a series of one or more "changes" to be made to the *web_file*. Each change in the change file includes the tokens tx, ty, and tz. The token tx signals a change. Between the tx and ty is the exact code from the *web_file* which is to be replaced by the code between the ty and tz from the CHange file.

9. Output of the program will be written to *met_file*. It includes:

1. The number of identifier tokens and number tokens recognized;
2. Software Science operators used, and their frequencies;
3. Software Science operands, and their frequences;
4. WEB program counts: total numbered code sections, number of procedures, and number of functions;
5. Size metrics: number of lines of limbo (lol), number of lines of documentation (lod), average number of lines of documentation per code section, number of lines of macro and format definitions (lom), number of lines of code (loc), and average number of lines of code per code section which had code;
6. Cyclomatic complexity number : vg;
7. Software Science measures: η_1, η_2, N_1, N_2, length, volume, effort, and time (in seconds, minutes, and hours); and
8. WEB environment specific commands and their frequencies.

As mentioned before, if **stat** is true, the output will be sent to the terminal in condensed form. This can be used as input to a statistics package with minimal modifications.

10. Halstead's Software Science. Halstead developed the Software Science family of measures which are calculated from four basic counts.

η_1 = the number of unique operators
η_2 = the number of unique operands
N_1 = the total number of occurrences of all operators
N_2 = the total number of occurrences of all operands

An operator is defined to be either a built-in function, a symbol or group of symbols that produce an action. An operand can be a constant or a variable.

The operators and operands to be considered will be clearly defined in the Pascal and WEB Counting Strategy sections.

The Software Science metrics to be calculated in the program are briefly described here.

1. *length* of a program in tokens N

$$N = N_1 + N_2$$

2. *volume* of a program V

The volume of a program is the fewest number of binary digits or bits with which the program can be represented.

$$V = N * log_2(\eta_1 + \eta_2)$$

3. *effort* E

The *effort* is the total number of elementary mental discriminations needed to write a program. The approximation of E, $\hat{E}$, does not need V^* (potential volume) for its calculation, and will be used.

$$E = \frac{v^2}{V^*}$$

$$\hat{E} = V * [\frac{(\eta_1 * N_2)}{(2 * \eta_2)}]$$

4. *time* T

The *time* is *effort* converted to units of time based on the Stroud number, $S = 18$ e.m.d. per second.

$$T = \frac{E}{S}$$

⟨Procedure halstead 10⟩ ≡
procedure *halstead*(*eta1*, *eta2*, *n1*, *n2* : *integer*; **var** *hlength*, *hvolume*, *heffort*, *htime* : *real*);
  const *hstroud* = 18; { Stroud number }
  begin *hlength* ← *n1* + *n2*;
  if ((*eta1* + *eta2*) > 0) **then** *hvolume* ← *hlength* ∗ *ln*(*eta1* + *eta2*)/*ln*(2);
  if (*eta2* > 0) **then** *heffort* ← *hvolume* ∗ ((*eta1* ∗ *n2*)/(2 ∗ *eta2*));
  htime ← *heffort*/*hstroud*;
  end;

This code is used in section 2.

11. ⟨Calculate software science measures 11⟩ ≡
  ⟨Find $\eta 1$, $\eta 2$, N_1, and N_2 12⟩;
  halstead(*eta1*, *eta2*, *n1*, *n2*, *hlength*, *hvolume*, *heffort*, *htime*)

This code is used in section 3.

12. ⟨Find $\eta 1$, $\eta 2$, N_1, and N_2 12⟩ ≡

```
eta1 ← 0; eta2 ← 0; n1 ← 0; n2 ← 0;  { Pascal operators }
for l ← tand to tcolon do
  begin sym_string(l, s);
  if (s ≠ '␣␣␣␣␣␣␣␣␣␣') then
    if (count[l] ≠ 0) then
      begin incr(eta1); n1 ← n1 + count[l];
      end;
  end;  { User-defined subprograms and macros }
for i ← 1 to numuser do
  begin incr(eta1); n1 ← n1 + usercnt[i];
  end;  { Operands }
for i ← 1 to numopd do
  if (opdcnt[i] ≠ 0) then
    begin incr(eta2); n2 ← n2 + opdcnt[i];
    end;
```

This code is used in section 11.

13. ⟨Global variables of the program 13⟩ ≡

```
eta1,  { η1 - number of unique operators }
eta2,  { η2 - number of unique operands }
n1,  { N1 - total number of occurrences of all operators }
n2  { N2 - total number of occurrences of all operands }
: integer; hlength,  { length of a program in tokens }
hvolume,  { volume of a program in bits }
heffort,  { estimated effort needed to write a program }
htime  { effort converted to time using the Stroud number }
: real;
```

See also sections 16, 20, 62, and 80.

This code is used in section 2.

14. McCabe's Cyclomatic Complexity. McCabe's cyclomatic complexity metric measures program control flow complexity. The easiest way to compute *vg* is to count the number of conditions (predicates) in a program and add one. The "add one" is accounted for by counting the **program** reserved word. See the Pascal Counting Strategy for more detail.

```
⟨ Procedure mccabe 14 ⟩ ≡
procedure mccabe(sym : token; var vg : integer);
  var temp: integer;
  begin temp ← vg;   { for debug }
  if (sym ∈ [twhile, tif, tuntil]) then
    begin condition ← true; incr(vg);
    end
  else if (sym ∈ [tfor, tprocedure, tfunction, tprogram]) then incr(vg)
    else if condition ∧ (sym ∈ [tand, tor]) then incr(vg)
      else if sym = tcase then decr(casebal)
        else if (casebal ≥ 1) ∧ (parenbal = 0) ∧ (sym ∈ [tcolon, tcomma]) then incr(vg)
          else if (sym = tbegin) ∧ (casebal ≥ 1) then incr(casebal)
            else if (sym = tend) ∧ (casebal ≥ 1) then decr(casebal)
              else if condition ∧ (sym ∈ [tdo, tthen, tsemi]) then condition ← false
                else if (sym = tlabel) then decr(vg)
                  else if islabel ∧ (sym = tcomma) then decr(vg);
  debug if (temp ≠ vg) then   { changed, so print }
    write('␣VG␣', vg : 1, '␣');
  gubed
  end;
```

This code is used in section 2.

15. *mccabe* is called by the parser.

```
⟨ Update cyclomatic complexity 15 ⟩ ≡
  mccabe(sym, vg)
```

This code is used in section 67.

16. ⟨ Global variables of the program 13 ⟩ +≡

```
vg: integer;   { McCabe's cyclomatic complexity number }
condition: boolean;   { current statement is a conditional }
casebal: integer;   { add one if tcase, subtract one if tend }
```

17. Pascal Counting Strategy. This counting strategy is adapted from S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models.* Menlo Park, CA: Benjamin/Cummings, 1986, pp. 38–41.

1. All of the program code including statement parts, program heading, and declaration parts should be considered.
2. Variables, constants (including the standard constants **FALSE**, **TRUE**, and **MAXINT**), user-defined types, literals, file names, and the reserved word **NIL** are counted as operands. All operands are counted as if they were global in scope. In other words, local variables with the same name in different procedures are counted as multiple occurrences of the same operand.
3. The following entities are always counted as single operators (* is not differentiated between set and arithmetic use):

*	/	DIV	MOD	<	<=	;
<>	>=	>	:=	^	,	..
NOT	AND	OR	IN	PACKED	TO	DOWNTO
INTEGER	REAL	TEXT	CHAR	LABEL	PROGRAM	FUNCTION
FORWARD	READ	READLN	WRITE	WRITELN	BOOLEAN	
PROCEDURE		OTHERWISE		EXTERNAL		

4. The following multiple entities are counted as single operators.

BEGIN END	CASE END	WHILE DO	REPEAT UNTIL
IF THEN	IF THEN ELSE	FOR DO	WITH DO
SET OF	FILE OF	RECORD END	ARRAY OF

5. The following entities or pairs of entities are counted as single operators subject to the accompanying conditions:

VAR	is counted as an operator in parameter lists and is not counted as a section label
=	is counted as either a relational operator in expressions or a definition operator in non-executable sections of the program.
+	is counted as either a unary + or binary + depending on its function. The binary - is not differentiated between arithmetic and set usage.
.	is counted as either a record component selector symbol or a program terminator depending on its function.
:	is a definition operator in the **VAR** section and parameter lists. It is a separation operator following **CASE** or **GOTO** labels.
( )	is counted as either an argument list operator or expression operator depending on the function.
[ ]	is counted as either a subscript operator or set operator depending on the function.

6. Procedure and function calls are counted as operators. The subprogram name following **FUNCTION** or **PROCEDURE** is not counted, though it actually is the operand for the **FUNCTION** or **PROCEDURE** operator.
7. **GOTO** statements (i.e., **GOTO** and an accompanying label) are counted as the operator **GOTO** and the operand *label*.

8. Declarations of labels are not enumerated — all tokens after the **LABEL** operator through the next semicolon (inclusive) are ignored.
9. The following are syntactic devices and are not counted:

   CONST **TYPE** **VAR** (for variable sections)

10. not applicable.
11. McCabe's **vg** metric is counted as follows:

    increment on keywords:

WHILE	**FOR**	**REPEAT**	**IF**
AND	**OR**	**PROCEDURE**	**FUNCTION**
PROGRAM			

AND and **OR** include loops/branches controlled by boolean variables. Conditionals could not be counted directly as boolean functions and variables would not be counted correctly.

also count **CASE** labels by :

incrementing on:

colons in the executable part of the program but outside a **WRITE(LN)** parameter list.

commas in a **CASE** label list.

decrementing on:

LABEL keyword.

commas in a **LABEL** statement.

The decrement is necessary to remove the **GOTO** labels, if any.

18. WEB Counting Strategy. The following WEB entities are counted and output to the *met_file*.

1. **lol** — *lines of limbo* : the number of lines of text before the first starred module is found.
2. **lod** — *lines of documentation* : the number of lines of TEXtext in the code sections of the program.
3. *average lines of documentation* : the average number of lines of TEX text in the code sections per code section.
4. **lom** — *lines of macros* : the number of lines of macro and format definitions in the code sections of the program.
5. **loc** — *lines of code* : the number of lines of program code in the code sections of the program. This includes in-line comments. Blank lines are not counted.
5. *average lines of code* : the average number of lines of program code in the code sections of the program which have code. Note that this may be different from the total number of code sections in the program used to compute lod_cs.
6. The count of each WEB token used in the program is output to *metfile*. (See procedure sym_string or function web_token for the complete list.)
7. Macro calls are counted as operators. The identifier name following the macro is not counted.

19. Data Structures. Tokens to be scanned will include four categories.

1. Pascal identifiers and numbers;
2. Pascal reserved words, predefined data types, standard Pascal functions and procedures, arithmetic, set and relational operators.
3. Special tokens unique to WEB and Pascal; and
4. WEB commands.

Blank lines will be skipped.

Each token is declared in the enumerated type *token*. An array *count* will hold the number of occurrences of each token. It may be more convenient to split up the various token classes (operand, operator) later, for more detailed metric calculations.

In addition, there is an array *pword* which holds all the Pascal reserved words, etc (see item 2 above). This will be used to check if strings are actually reserved words or not. A parallel array *psymbol* holds the actual token name for each string.

⟨Global types of the program 19⟩ ≡
token = (*tnul*, *tident*, *tnum*,
tand, *tarray*, *tbegin*, *tboolean*, *tcase*, *tchar*, *tconst*, *tdiv*, *tdo*, *tdownto*, *telse*, *tend*, *teof*, *teoln*, *texternal*, *tfile*, *tfor*, *tforward*, *tfunction*, *tgoto*, *tif*, *tin*, *tinteger*, *tlabel*, *tmod*, *tnot*, *tof*, *tor*, *totherwise*, *tpacked*, *tprocedure*, *tprogram*, *tread*, *treadln*, *treal*, *trecord*, *trepeat*, *tset*, *ttext*, *tthen*, *tto*, *ttype*, *tuntil*, *tvar*, *twhile*, *twith*, *twrite*, *twriteln*,
tplus, *tminus*, *ttimes*, *tslash*, *tlparen*, *trparen*, *teql*, *tcomma*, *tperiod*, *tlt*, *tgt*, *tsemi*, *tlbrack*, *trbrack*, *tbecomes*, *tdotdot*, *tneq*, *tleq*, *tgeq*, *tcaret*, *tcolon*,
tassumpt, *tstring*, *tpreproc*, *tcomment*, *targument*, *tdbl_eql*, *tone_eql*, *tcall*, *tis*, *tabbrev*,
tat, *tnew_mod*, *tstar_mod*, *tdef*, *tformat*, *tbegin_code*, *tmod_name*, *tend_web*, *toctal*, *thex*, *tcheck_sum*, *tbegin_comment*, *tend_comment*, *tjoin*, *troman*, *ttypewriter*, *tuser_def*, *ttex_string*, *tverbatim*, *tforce_line*, *tunderline*, *tno_underline*, *tthin_space*, *tline_break*, *topt_line_break*, *tbig_line_break*, *tno_line_break*, *tpseudo_semi*, *tx*, *ty*, *tz*);
alpha = **array** [1 .. *len_alpha*] **of** *char*;
alphat = **array** [1 .. *num_rw*] **of** *alpha*;
tokent = **array** [1 .. *num_rw*] **of** *token*;
symbolt = **array** [*token*] **of** *integer*;

See also sections 61 and 78.

This code is used in section 2.

20. ⟨Global variables of the program 13⟩ +≡
pword: *alphat*; { strings of reserved words }
psymbol: *tokent*; { tokens of reserved words }
count: *symbolt*; { total num. of occurrences of each token }

21. ⟨Global constants of the program 21⟩ ≡
num_rw = 48; { num. of Pascal reserved words }
len_alpha = 10; { max. length of an identifier }

See also sections 63 and 79.

This code is used in section 2.

22. Each pascal reserved word will be a token itself. They will be stored in an array *pword*. The token name corresponding to each reserved word will be located in the array *psymbol*.

⟨Initialize arrays 22⟩ ≡

```
pword[1] ← 'AND␣␣␣␣␣␣␣'; pword[2] ← 'ARRAY␣␣␣␣␣'; pword[3] ← 'BEGIN␣␣␣␣␣';
pword[4] ← 'BOOLEAN␣␣␣'; pword[5] ← 'CASE␣␣␣␣␣␣'; pword[6] ← 'CHAR␣␣␣␣␣␣';
pword[7] ← 'CONST␣␣␣␣␣'; pword[8] ← 'DIV␣␣␣␣␣␣␣'; pword[9] ← 'DO␣␣␣␣␣␣␣␣';
pword[10] ← 'DOWNTO␣␣␣␣'; pword[11] ← 'ELSE␣␣␣␣␣␣'; pword[12] ← 'END␣␣␣␣␣␣␣';
pword[13] ← 'EOF␣␣␣␣␣␣␣'; pword[14] ← 'EOLN␣␣␣␣␣␣'; pword[15] ← 'EXTERNAL␣␣';
pword[16] ← 'FILE␣␣␣␣␣␣'; pword[17] ← 'FOR␣␣␣␣␣␣␣'; pword[18] ← 'FORWARD␣␣␣';
pword[19] ← 'FUNCTION␣␣'; pword[20] ← 'GOTO␣␣␣␣␣␣'; pword[21] ← 'IF␣␣␣␣␣␣␣␣';
pword[22] ← 'IN␣␣␣␣␣␣␣␣'; pword[23] ← 'INTEGER␣␣␣'; pword[24] ← 'LABEL␣␣␣␣␣';
pword[25] ← 'MOD␣␣␣␣␣␣␣'; pword[26] ← 'NOT␣␣␣␣␣␣␣'; pword[27] ← 'OF␣␣␣␣␣␣␣␣';
pword[28] ← 'OR␣␣␣␣␣␣␣␣'; pword[29] ← 'OTHERWISE␣'; pword[30] ← 'PACKED␣␣␣␣';
pword[31] ← 'PROCEDURE␣'; pword[32] ← 'PROGRAM␣␣␣'; pword[33] ← 'READ␣␣␣␣␣␣';
pword[34] ← 'READLN␣␣␣␣'; pword[35] ← 'REAL␣␣␣␣␣␣'; pword[36] ← 'RECORD␣␣␣␣';
pword[37] ← 'REPEAT␣␣␣␣'; pword[38] ← 'SET␣␣␣␣␣␣␣'; pword[39] ← 'TEXT␣␣␣␣␣␣';
pword[40] ← 'THEN␣␣␣␣␣␣'; pword[41] ← 'TO␣␣␣␣␣␣␣␣'; pword[42] ← 'TYPE␣␣␣␣␣␣';
pword[43] ← 'UNTIL␣␣␣␣␣'; pword[44] ← 'VAR␣␣␣␣␣␣␣'; pword[45] ← 'WHILE␣␣␣␣␣';
pword[46] ← 'WITH␣␣␣␣␣␣'; pword[47] ← 'WRITE␣␣␣␣␣'; pword[48] ← 'WRITELN␣␣␣';
psymbol[1] ← tand; psymbol[2] ← tarray; psymbol[3] ← tbegin; psymbol[4] ← tboolean;
psymbol[5] ← tcase; psymbol[6] ← tchar; psymbol[7] ← tconst; psymbol[8] ← tdiv; psymbol[9] ← tdo;
psymbol[10] ← tdownto; psymbol[11] ← telse; psymbol[12] ← tend; psymbol[13] ← teof;
psymbol[14] ← teoln; psymbol[15] ← texternal; psymbol[16] ← tfile; psymbol[17] ← tfor;
psymbol[18] ← tforward; psymbol[19] ← tfunction; psymbol[20] ← tgoto; psymbol[21] ← tif;
psymbol[22] ← tin; psymbol[23] ← tinteger; psymbol[24] ← tlabel; psymbol[25] ← tmod;
psymbol[26] ← tnot; psymbol[27] ← tof; psymbol[28] ← tor; psymbol[29] ← totherwise;
psymbol[30] ← tpacked; psymbol[31] ← tprocedure; psymbol[32] ← tprogram; psymbol[33] ← tread;
psymbol[34] ← treadln; psymbol[35] ← treal; psymbol[36] ← trecord; psymbol[37] ← trepeat;
psymbol[38] ← tset; psymbol[39] ← ttext; psymbol[40] ← tthen; psymbol[41] ← tto;
psymbol[42] ← ttype; psymbol[43] ← tuntil; psymbol[44] ← tvar; psymbol[45] ← twhile;
psymbol[46] ← twith; psymbol[47] ← twrite; psymbol[48] ← twriteln;
for sym ← tnul to tpseudo_semi do count[sym] ← 0;
```

This code is used in section 3.

23. Lexical Analysis. Lexical Analysis will be handled depending on which part of a WEB program we are currently in. The very first part of a WEB program includes several lines of "limbo" which may include TEX commands, or general comments about the program which will not appear in the WEAVEd output. The rest of the program is made of WEB modules. A WEB module is recognized by the tokens *tstar_mod* or *tnew_mod*. Each module may have up to three parts as follows (any may be empty, but they must appear in this order): a TEX part, explaining what the module does; a definition part, defining the macros to be used in the Pascal code; and finally, the Pascal section, which contains the Pascal code.

Thus, there are four plus two extra states to be handled here. The tokens which cause the state to change to these states are listed below.

1. Limbo section – start state
2. TEX section – *tnew_mod*, *tstar_mod*
3. Definition section – *tdef*, *tdef*
4. Pascal section – *tbegin_code*, *tmod_name*
5. Finished – end of input – no more tokens
6. Parsing – any valid token recognized by Pascal

The following macros will be used to identify different states.

define *same_state* = 0
define *limbo_state* = 1
define *tex_state* = 2
define *def_state* = 3
define *code_state* = 4
define *fin_state* = 5
define *parse_state* = 6

24. Basically, when in states limbo, tex, or def, the input will be scanned, counting all WEB commands and other tokens of interest (by incrementing the appropriate symbols in *count*), until a state change occurs. When in the code state, the input will be scanned until a token is found. If the token is of interest to the parser, the state will change to the parse state, with *sym* holding the token; otherwise the state will stay the same. Either way, all tokens are counted (with some exceptions documented fully).

In addition to recognizing tokens, the lexical analyzer will keep primitive counts of the lines of limbo, lines of documentation, number of macros, and lines of code.

```
⟨Perform lexical analysis 24⟩ ≡
  state ← limbo_state;
  while state ≠ fin_state do
    begin case state of
    limbo_state: next ← proc_limbo;
    tex_state: next ← proc_tex;
    def_state: next ← proc_def;
    code_state: next ← proc_code;
    parse_state: next ← parser;
    othercases do_nothing
    end; ⟨Adjust counts of lol, lod, lom, loc 37⟩;
    prev_state ← state; state ← next;
    debug write('(', prev_state : 1, state : 1, ')␣');
    gubed
    end;
```

This code is used in section 3.

25. These algorithms are adapted from Knuth's WEAVE program, and Niklaus Wirth's book titled *Algorithms + Data Structures = Programs.*

⟨ Procedures for lexical analysis 25 ⟩ ≡
  ⟨ Function proc_limbo return *next* 27 ⟩;
  ⟨ Function proc_tex return *next* 28 ⟩;
  ⟨ Function proc_def return *next* 29 ⟩;
  ⟨ Function proc_code return *next* 30 ⟩;

See also section 26.

This code is used in section 2.

26. The parser is called in the lexical analysis phase. See major section on Parsing.

⟨ Procedures for lexical analysis 25 ⟩ +≡
  ⟨ Utilities of parser 77 ⟩
  ⟨ Function parser return *next* 67 ⟩;

27. Function proc_limbo skips through the input file counting the number of lines, until a new module is found (either token *tnew_mod* or *tstar_mod* is recognized). It will return *tex_state*.

⟨ Function proc_limbo return *next* 27 ⟩ ≡

```
function proc_limbo: integer;
  var ret_state: integer;
  begin ret_state ← same_state;
  while (ret_state = same_state) do
    begin ⟨ if end of buffer, get new buffer 31 ⟩;
    if end_of_input then ret_state ← fin_state
    else begin sym ← tnul; buffer[lbuf + 1] ← '@';
      while (buffer[cbuf] ≠ '@') do incr(cbuf);
      if (cbuf ≤ lbuf) then
        begin cbuf ← cbuf + 2; cc ← buffer[cbuf − 1]; sym ← web_token(cc); incr(count[sym]);
        debug outsym(sym);
        gubed
        if sym ∈ [tnew_mod, tstar_mod] then ret_state ← tex_state;
        end;
      end;
    end;
  proc_limbo ← ret_state;
  end
```

This code is used in section 25.

28. Function proc_tex skips through the TEX text, counting each WEB command and Pascal assumption (delimited by two tassumpt symbols). State change will be flagged when recognizing one of the following tokens: *tnew_mod*, *tstar_mod*, *tdef*, *tformat*, *tbegin_code*, and *tmod_name*.

```
⟨Function proc_tex return next 28⟩ ≡
function proc_tex: integer;
  var ret_state, bal: integer;
  begin ret_state ← same_state;
  while (ret_state = same_state) do
    begin ⟨if end of buffer, get new buffer 31⟩;
    if end_of_input then ret_state ← fin_state
    else begin sym ← tnul; buffer[lbuf + 1] ← '@';
      repeat cc ← buffer[cbuf]; incr(cbuf);
      until (cc = '@') ∨ (cc = '|');
      if (cc = '|') then ⟨Process an assumption 38⟩
      else if (cbuf ≤ lbuf) then ⟨Get a web command 39⟩;
      end;
    if sym ≠ tnul then
      begin incr(count[sym]);
      debug outsym(sym);
      gubed
      end;
    end;
  proc_tex ← ret_state;
  end  { function }
```

This code is used in section 25.

29. Function proc_def basically parses part of the macro or format statement and then passes control to the code state. The macro name is placed in the *user* array, but is not counted as an operator (the same policy as for function and procedure names). Tokens are counted up to and including the '=' or '==', when control passes to the code_state which processes the following Pascal code or constant.

Format statements are only counted for their tokens *tident*, *tdbl_eql*, and *tident*. The identifiers are not saved or counted as operators or operands because they are used by `WEAVE` for formatting.

The syntax of macros and format statements follow.

1. Numeric macro : identifier = constant
2. Simple macro : identifier == Pascal code
3. Parametric macro : identifier (argument) == Pascal code
4. Format definition: identifier == identifier

```
⟨Function proc_def return next 29⟩ ≡
function proc_def: integer;
  var ret_state, j, k: integer;
  begin ret_state ← same_state;
  while (ret_state = same_state) do
    begin ⟨if end of buffer, get new buffer 31⟩;
    if end_of_input then ret_state ← fin_state
    else begin buffer[lbuf + 1] ← '@'; ⟨Skip blanks in buffer 42⟩;
      ⟨Get an identifier 48⟩;
      if sym = tdef then
        begin sym ← tident; incr(count[sym]);
        debug outsym(sym);
        gubed⟨Skip blanks in buffer 42⟩;
        ⟨Save user-defined name in user 74⟩;
        if cc = '=' then
          if buffer[cbuf] = '=' then ⟨Process simple macro 43⟩
          else ⟨Process numeric macro 44⟩
        else if cc = '(' then ⟨Process parametric macro 45⟩;
        end
      else if sym = tformat then ⟨Process format definition 46⟩;
      end;
    end;
  proc_def ← ret_state;
  end
```

This code is used in section 25.

30. Function proc_code sets *sym* to the first token it finds. If it is an insignificant `WEB` command (i.e., if it doesn't have an effect on the Pascal code) or a Pascal comment, the token is counted but not sent to the parser. Otherwise, all tokens are passed to the parser after being counted, unless it is a `WEB` command indicating that a different state change is to be made. Exceptions are :

1. The syntactic device *tvar* is not counted, and is left to the parser to deal with because it is counted only if in a parameter list;
2. *tconst* and *ttype* are not counted as they are also syntactic devices;
3. the following tokens are the second half of a pair of reserved words counted as one operator, and thus will not be counted (ever): *tof*, *tend*, *tdo*, *tthen*, *tuntil*, *trparen*, and *trbrack*.

```
⟨Function proc_code return next 30⟩ ≡
function proc_code: integer;
  var ret_state, k, i, j: integer;
  begin ret_state ← same_state;
  while ret_state = same_state do
    begin ⟨if end of buffer, get new buffer 31⟩;
    if end_of_input then ret_state ← fin_state
    else begin sym ← tnul; buffer[lbuf + 1] ← ´@´; ⟨Skip blanks in buffer 42⟩;
          { assume return state unless otherwise set }
      ret_state ← parse_state;
      if cc ∈ [´A´ .. ´Z´, ´a´ .. ´z´] then
        begin ⟨Get an identifier 48⟩;
        ⟨Check if a reserved word 49⟩;
        end
      else if cc ∈ [´0´ .. ´9´] then ⟨Get a number 50⟩
        else if (cc = ´@´) ∧ (cbuf ≤ lbuf) then ⟨Get a web command 39⟩
          else if (cc = ´´´´) then ⟨Get a constant string 51⟩
            else if (cc = ´"´) then ⟨Get a preprocessed string 52⟩
              else if (cc = ´{´) then
                  begin ⟨Get a pascal comment 53⟩;
                  ret_state ← same_state;
                  end
                else if (cc = ´#´) then ⟨Get a macro argument 54⟩
                  else if (cbuf ≤ lbuf + 1) then ⟨Get an operator 55⟩;  { if }
      if ¬(sym ∈ [tvar, tconst, ttype, tof, tend, tdo, tthen, tuntil, trparen, trbrack]) then incr(count[sym]);
      debug outsym(sym);
      gubed
      end;
    end;
  proc_code ← ret_state;
  end
```

This code is used in section 25.

31. This section of code returns a new *buffer* if the last character in the *buffer* has been read, by calling procedure *get_line*. The line or number counts *lol*, *lod*, *loc*, *lom* are incremented accordingly.

```
⟨if end of buffer, get new buffer 31⟩ ≡
  if (cbuf > lbuf) then
    begin get_line;
    case state of
    limbo_state: begin incr(lol); ⟨Output lol 32⟩;
      end;
    tex_state: begin incr(lod); ⟨Output lod 33⟩;
      end;
    def_state: begin incr(lom); ⟨Output lom 34⟩;
      end;
    code_state: begin incr(loc); ⟨Output loc 35⟩;
      end;
      otherwise do_nothing;
    end; {case}
    ⟨Output eoln 36⟩;
    end
```

This code is used in sections 27, 28, 29, 30, 40, and 41.

32. Print to the sceen if debug is on.

```
⟨Output lol 32⟩ ≡
  debug write('␣l=', lol : 1, '␣');
  gubed
```

This code is used in sections 31 and 37.

33.

```
⟨Output lod 33⟩ ≡
  debug write('␣D=', lod : 1, '␣');
  gubed
```

This code is used in sections 31, 37, 37, 37, 37, and 37.

34.

```
⟨Output lom 34⟩ ≡
  debug write('␣M=', lom : 1, '␣');
  gubed
```

This code is used in sections 31, 37, and 37.

35.

```
⟨Output loc 35⟩ ≡
  debug write('␣C=', loc : 1, '␣');
  gubed
```

This code is used in sections 31, 37, 37, and 37.

36.

```
⟨Output eoln 36⟩ ≡
  debug writeln;
  gubed
```

This code is used in section 31.

37. Since the *lol*, *lod*, *lom*, *loc* counts are incremented in the state which reads the new *buffer*, we must adjust the counts if the pending change of the state indicates that the *buffer* line really belongs to another state. For example, if in *code_state* we read a new *buffer* starting with the token *tdef*, we know that the *loc* was incremented prematurely. To adjust the counts, we will then decrement *loc* and increment *lom*.

⟨Adjust counts of *lol*, *lod*, *lom*, *loc* 37⟩ ≡

```
begin if state = limbo_state then
  begin if next = tex_state then
    begin incr(lod); decr(lol); ⟨Output lol 32⟩;
    ⟨Output lod 33⟩;
    end;
  end
else if state = tex_state then
    begin if next = def_state then
      begin incr(lom); decr(lod); ⟨Output lod 33⟩;
      ⟨Output lom 34⟩;
      end
    else if next = code_state then
        begin incr(loc); decr(lod); ⟨Output lod 33⟩;
        ⟨Output loc 35⟩;
        end
      else if next = fin_state then
          begin decr(lod); ⟨Output lod 33⟩;
          end
    end
  else if state = code_state then
      begin if next = def_state then
        begin incr(lom); decr(loc); ⟨Output loc 35⟩;
        ⟨Output lom 34⟩;
        end
      else if next = tex_state then
          begin incr(lod); decr(loc); ⟨Output loc 35⟩;
          ⟨Output lod 33⟩;
          end;
      end;
end
```

This code is used in section 24.

38. When using two tassumpt symbols as delimiters in a WEB program, WEAVE formats the characters in between as if it were a Pascal identifier. Thus, it may be useful to count such occurrences as an "assumption" of some kind. Examples: a, b, c will be counted as three assumptions; $a[i, j]$ will be counted as one.

```
⟨Process an assumption 38⟩ ≡
  begin sym ← tassumpt; bal ← 0;
  repeat cc ← buffer[cbuf]; incr(cbuf);
    if (cc = ´,´) ∧ (bal = 0) then
      begin incr(count[sym]);
      debug outsym(sym);
      gubed
      end
    else if (cc = ´[´) then incr(bal)
      else if (cc = ´]´) then decr(bal);
  until (cc = ´|´); {second delimeter}
  end
```

This code is used in section 28.

39. If the WEB token is *tnew_mod*, *tstar_mod*, *tdef*, *tformat*, *tbegin_code*, or *tmod_name*, a state change will be flagged. Otherwise the state will remain the same. If the token is *tmod_name*, *troman*, *ttypewriter*, *tuser_def*, *ttex_string*, or *tverbatim*, the input will be skipped until the token *tend_web* is found. Likewise, if *tbegin_comment* is recognized, a skip to *tend_comment* will be made. Finally if the token *tforce_line* is found, the count *loc* will be incremented.

NOTE: the *count* array is not explicitly incremented here.

```
⟨Get a web command 39⟩ ≡
  begin cc ← buffer[cbuf]; incr(cbuf); sym ← web_token(cc);
  if sym ∈ [tnew_mod, tstar_mod, tdef, tformat, tbegin_code, tmod_name] then
    begin case sym of
    tnew_mod, tstar_mod: ret_state ← tex_state;
    tdef, tformat: ret_state ← def_state;
    tbegin_code: ret_state ← code_state;
    tmod_name: begin ⟨Skip text to tend_web 40⟩;
      if state = tex_state then ret_state ← code_state
      else if state = code_state then ret_state ← parse_state;
      end;
    othercases do_nothing;
    end;
    end
  else begin ret_state ← same_state;
    if sym ∈ [troman, ttypewriter, tuser_def, ttex_string, tverbatim] then ⟨Skip text to tend_web 40⟩
    else if (sym = tbegin_comment) ∧ (prev_state ≠ def_state) then ⟨Skip text to tend_comment 41⟩
      else if sym = tforce_line then
          begin incr(loc);
          debug write(´␣C=´, loc : 1);
          gubed
          end;
    end;
  end
```

This code is used in sections 28 and 30.

40. This section skips the input until the token *tend_web* is encountered. Since this does not have to be on the same input line, a new *buffer* is read when necessary. If the current token is *tmod_name*, we check to see what it's purpose is: a placeholder of code or a call to a code section. If a semicolon is found after a call to a code section, it is counted as an operator; however, a pseudo-semicolon is not counted as a Pascal operator.

```
⟨Skip text to tend_web 40⟩ ≡
  begin repeat cc ← buffer[cbuf]; incr(cbuf); ⟨if end of buffer, get new buffer 31⟩;
    if cbuf = 1 then  { got a new buffer }
      begin buffer[lbuf + 1] ← '@'; cc ← buffer[cbuf]; incr(cbuf);
      end;
  until (cc = '@') ∧ (buffer[cbuf] = '>');
  cc ← buffer[cbuf]; incr(cbuf);  {read the > sign}
  incr(count[tend_web]);
  if sym = tmod_name then
    begin incr(count[sym]); cc ← buffer[cbuf]; incr(cbuf);
    if cc = '=' then  {a web module definition, placeholder}
      sym ← tis
    else  {a web module call}
    begin sym ← tcall;
    if (cc = '@') ∧ (buffer[cbuf] = ';') then  {this is a psuedosemi}
      begin incr(cbuf); incr(count[tpseudo_semi]);
      end
    else if (cc = ';') then  {count tsemi next time around}
        decr(cbuf);
    end;
    end;
  end
```

This code is used in sections 39 and 39.

41. This section skips the input until the token *tend_comment* is encountered. Since this does not have to be on the same input line, a new *buffer* is read when necessary.

NOTE: the usage of the tokens *tbegin_comment*, and *tend_comment* allows code to be "commented out" where Pascal does not allow it (mainly because you can't nest comment symbols).

```
⟨Skip text to tend_comment 41⟩ ≡
  begin repeat cc ← buffer[cbuf]; incr(cbuf); ⟨if end of buffer, get new buffer 31⟩;
    if cbuf = 1 then  {got a new buffer}
      begin buffer[lbuf + 1] ← '@'; cc ← buffer[cbuf]; incr(cbuf);
      end;
  until (cc = '@') ∧ (buffer[cbuf] = '}');
  cc ← buffer[cbuf]; incr(cbuf); incr(count[tend_comment]);
  end
```

This code is used in section 39.

42. Skip blanks in *buffer*. The next valid character is *cc*.

```
⟨Skip blanks in buffer 42⟩ ≡
  repeat cc ← buffer[cbuf]; incr(cbuf);
  until (cc ≠ '␣');
```

This code is used in sections 29, 29, 30, 45, 45, 45, 46, and 46.

43. Parse '=='.

⟨ Process simple macro 43 ⟩ ≡
  begin *sym* ← *tdbl_eql*; *incr*(*count*[*sym*]);
  debug *outsym*(*sym*);
  gubed *cc* ← *buffer*[*cbuf*]; *incr*(*cbuf*); *ret_state* ← *code_state*;
  end

This code is used in section 29.

44. Parse '='.

⟨ Process numeric macro 44 ⟩ ≡
  begin *sym* ← *tone_eql*; *incr*(*count*[*sym*]);
  debug *outsym*(*sym*);
  gubed *ret_state* ← *code_state*;
  end

This code is used in section 29.

45. Parse '(*targument*) =='

⟨ Process parametric macro 45 ⟩ ≡
  begin *sym* ← *tlparen*; *incr*(*count*[*sym*]);
  debug *outsym*(*sym*);
  gubed ⟨ Skip blanks in *buffer* 42 ⟩;
  if $cc =$ '#' **then**
    begin *sym* ← *targument*; *incr*(*count*[*sym*]);
    debug *outsym*(*sym*);
    gubed ⟨ Skip blanks in *buffer* 42 ⟩;
    if $cc =$ ')' **then**
      begin *sym* ← *trparen*; *incr*(*count*[*sym*]);
      debug *outsym*(*sym*);
      gubed ⟨ Skip blanks in *buffer* 42 ⟩;
      if $(cc = \text{'='}) \land (buffer[cbuf] = \text{'='})$ **then**
        begin *sym* ← *tdbl_eql*; *incr*(*count*[*sym*]);
        debug *outsym*(*sym*);
        gubed *cc* ← *buffer*[*cbuf*]; *incr*(*cbuf*); *ret_state* ← *code_state*;
        end;
      end;
    end;
  end

This code is used in section 29.

46. Parse '== *tident*'.

```
⟨Process format definition 46⟩ ≡
  begin sym ← tident; incr(count[sym]);
  debug outsym(sym);
  gubed⟨Skip blanks in buffer 42⟩;
  if (cc = ´=´) ∧ (buffer[cbuf] = ´=´) then
    begin sym ← tdbl_eql; incr(count[sym]);
    debug outsym(sym);
    gubedcc ← buffer[cbuf]; incr(cbuf); ⟨Skip blanks in buffer 42⟩;
    ⟨Get an identifier 48⟩;
    sym ← tident; incr(count[sym]);
    debug outsym(sym);
    gubedret_state ← code_state;
    end;
  end
```

This code is used in section 29.

47. Declare a macro to set *id* to blanks. The parameter is the loop control variable, and is initialized to zero.

```
  define id_to_blanks(#) ≡
          for # ← 1 to len_alpha do id[#] ← ´␣´;
        # ← 0
```

48. An identifier can be up to *len_alpha* long. Each character is set to upper case and all '_' (underscore) characters are deleted.

```
⟨Get an identifier 48⟩ ≡
  begin id_to_blanks(k);
  repeat if (k < len_alpha) ∧ (cc ≠ ´_´) then
      begin incr(k); id[k] ← ch_upper(cc);
      end;
    cc ← buffer[cbuf]; incr(cbuf);
  until ¬(cc ∈ [´A´ .. ´Z´, ´a´ .. ´z´, ´0´ .. ´9´, ´_´]);
  decr(cbuf);
  end
```

This code is used in sections 29, 30, and 46.

49. Binary search the *pword* array to see if *id* is there. If it is, set *sym* to its counterpart in *psymbol*. If it is not there, *sym* is set to *tident*.

```
⟨Check if a reserved word 49⟩ ≡
  begin i ← 1; j ← num_rw;
  repeat k ← (i + j) div 2;
    if id ≤ pword[k] then j ← k − 1;
    if id ≥ pword[k] then i ← k + 1;
  until (i > j) ∨ (pword[k] = id);
  if (pword[k] = id) then sym ← psymbol[k]
  else sym ← tident;
  end
```

This code is used in section 30.

50. This section gets a number, and copies the string into the global variable *num*. It does not distinguish between integer or reals. If *sign* = *true* (set in the parser), the sign will be copied from *id*[1]. Since we are assuming that this is valid Pascal code, we simply copy everything until a non-valid character (it is not a digit, decimal point, or exponent) is found. Since a decimal point also appears in the token *tdotdot*, we check for this occurrence and backtrack if this is recognized. The maximum number of digits allowed is *len_alpha*.

```
⟨Get a number 50⟩ ≡
  begin sym ← tnum;
  for k ← 1 to len_alpha do num[k] ← '␣';
  k ← 0;
  if sign then
    begin incr(k);
    if prev_sym = tplus then num[k] ← '+'
    else num[k] ← '-';
    end;
  repeat if (cc = '.') ∧ (buffer[cbuf] = '.') then   { this is tdotdot, stop }
      cc ← '␣'
    else begin if (k ≤ len_alpha − 1) then
        begin incr(k); num[k] ← cc;
        end;
      cc ← buffer[cbuf]; incr(cbuf);
      end;
  until ¬(cc ∈ ['0' .. '9', '.', 'E']);
  decr(cbuf); cc ← buffer[cbuf − 1];
  end
```

This code is used in section 30.

51. This section skips the input until a close single quote is found. It is assumed that it must be found in the same *buffer*. The string is copied into *id*.

```
⟨Get a constant string 51⟩ ≡
  begin sym ← tstring; id_to_blanks(k);
  repeat if k < len_alpha − 1 then
      begin incr(k); id[k] ← cc;
      end;
    cc ← buffer[cbuf]; incr(cbuf);
  until (cc = '''');
  incr(k); id[k] ← cc;
  end
```

This code is used in section 30.

52. This section skips the input until a close double quote is found. It is assumed that it must be found in the same *buffer*. This is a WEB preprocessed string. It is copied into *id*.

```
⟨Get a preprocessed string 52⟩ ≡
  begin sym ← tpreproc; id_to_blanks(k);
  repeat if k < len_alpha − 1 then
      begin incr(k); id[k] ← cc;
      end;
    cc ← buffer[cbuf]; incr(cbuf);
  until (cc = '"');
  incr(k); id[k] ← cc;
  end
```

This code is used in section 30.

53. This section skips the input until a close brace is found. It is assumed it must be found in the same *buffer*. If there are nested braces, increment and decrement until balance is zero. Note: this is an in-line comment.

```
⟨Get a pascal comment 53⟩ ≡
  begin sym ← tcomment; bracebal ← 1;
  repeat cc ← buffer[cbuf]; incr(cbuf);
    if (cc = '{') then incr(bracebal)
    else if (cc = '}') then decr(bracebal);
  until (cc = '}') ∧ (bracebal = 0);
  end
```

This code is used in section 30.

54. This section simply sets the current token to *targument*.

```
⟨Get a macro argument 54⟩ ≡
  begin sym ← targument;
  end
```

This code is used in section 30.

55. Check if the token is a valid Pascal operator.

```
⟨Get an operator 55⟩ ≡
  begin case cc of
  '+': sym ← tplus;
  '-': sym ← tminus;
  '*': sym ← ttimes;
  '/': sym ← tslash;
  '(': sym ← tlparen;
  ')': sym ← trparen;
  '=': sym ← teql;
  ',': sym ← tcomma;
  ';': sym ← tsemi;
  '[': sym ← tlbrack;
  ']': sym ← trbrack;
  '^': sym ← tcaret;
  '.': ⟨Return '..' or '.' 56⟩;
  '<': ⟨Return < or <> or <= 57⟩;
  '>': ⟨Return > or >= 58⟩;
  ':': ⟨Return ':' or ':=' 59⟩;
  othercases debug writeln(cc);
    gubed
  end;
  end
```

This code is used in section 30.

56. Check if the token is *tdotdot* or *tperiod*.

```
⟨Return '..' or '.' 56⟩ ≡
  begin if (buffer[cbuf] = '.') then
    begin sym ← tdotdot; cc ← buffer[cbuf]; incr(cbuf);
    end
  else sym ← tperiod;
  end
```

This code is used in section 55.

57. Check if the token is *tlt*, *tneq*, or *tleq*.

```
⟨Return < or <> or <= 57⟩ ≡
  begin if (buffer[cbuf] = '=') then
    begin sym ← tleq; cc ← buffer[cbuf]; incr(cbuf);
    end
  else if (buffer[cbuf] = '>') then
      begin sym ← tneq; cc ← buffer[cbuf]; incr(cbuf);
      end
    else sym ← tlt;
  end
```

This code is used in section 55.

58. Check if the token is *tgt* or *tgeq*.

```
⟨Return > or >= 58⟩ ≡
  begin if (buffer[cbuf] = '=') then
    begin sym ← tgeq; cc ← buffer[cbuf]; incr(cbuf);
    end
  else sym ← tgt;
  end
```

This code is used in section 55.

59. Check if the token is *tbecomes* or *tcolon*.

```
⟨Return ':' or ':=' 59⟩ ≡
  begin if (buffer[cbuf] = '=') then
    begin sym ← tbecomes; cc ← buffer[cbuf]; incr(cbuf);
    end
  else sym ← tcolon;
  end
```

This code is used in section 55.

60. Utilities for Lexical Analysis. Before defining all the utilities used in the previous code sections, let's define and initialize the many global variables we have been using.

⟨ Utilities for lexical analysis 60 ⟩ ≡
  ⟨ Function web_token return *token* 65 ⟩;
  ⟨ Function ch_upper return *char* 66 ⟩;

This code is used in section 2.

61. ⟨ Global types of the program 19 ⟩ +≡
  buffert = **array** [1 .. *len_line*] **of** *char*;
  file_name = **packed array** [1 .. *len_name*] **of** *char*;

62. ⟨ Global variables of the program 13 ⟩ +≡
web_file, *met_file*: *text*;
in_file, *out_file*: *file_name*;
buffer: *buffert*; { buffered line of input file }
end_of_input: *boolean*; { true if *eof* }
cbuf: *integer*; { current character to read from *buffer* }
lbuf: *integer*; { num of characters in current *buffer* }
cc: *char*; { last character read from *buffer* }
sym: *token*; { last token recognized }
id: *alpha*; { last identifier read }
num: *alpha*; { last number read - a string }
lol: *integer*; { lines of limbo }
loc: *integer*; { lines of code metric }
lod: *integer*; { lines of documentation metric }
lom: *integer*; { number of macros }
i: *integer*; { loop variable }
s: *alpha*; { temporary variable }
l: *token*; { for loop counter }
state: *integer*; { current state of lexical analysis }
prev_state: *integer*; { previous state }
next: *integer*; { next state to go to }
bracebal: *integer*; { used to check for nested braces }

63. ⟨ Global constants of the program 21 ⟩ +≡
  len_name = 25; { max length of a file name }
  len_line = 81; { max length of input buffer }

64. ⟨ Initialize global variables 64 ⟩ ≡
  vg ← 0;
  for *i* ← 1 **to** *len_line* **do** *buffer*[*i*] ← ´␣´;
  end_of_input ← *false*; *lbuf* ← 0; *cbuf* ← 1; { set cbuf > lbuf }
  cc ← ´␣´; *sym* ← *tnul*; *id_to_blanks*(*i*);
  for *i* ← 1 **to** *len_alpha* **do** *num*[*i*] ← ´␣´;
  loc ← 0; *lod* ← 0; *lol* ← 0; *lom* ← 0; *bracebal* ← 0; *state* ← *same_state*; *next* ← *same_state*;
  prev_state ← *same_state*; ⟨ Open input and output files 85 ⟩

See also section 81.

This code is used in section 3.

65. Function web_token will return the WEB command being recognized. It is assumed an '@' has just been encountered, prior to the call.

```
⟨Function web_token return token 65⟩ ≡
function web_token(d : char): token;
  begin case d of
  '@': web_token ← tat;
  '␣': web_token ← tnew_mod;
  '*': web_token ← tstar_mod;
  'd', 'D': web_token ← tdef;
  'f', 'F': web_token ← tformat;
  'p', 'P': web_token ← tbegin_code;
  '<': web_token ← tmod_name;
  '>': web_token ← tend_web;
  '''': web_token ← toctal;
  '"': web_token ← thex;
  '$': web_token ← tcheck_sum;
  '{': web_token ← tbegin_comment;
  '}': web_token ← tend_comment;
  '&': web_token ← tjoin;
  '^': web_token ← troman;
  '.': web_token ← ttypewriter;
  ':': web_token ← tuser_def;
  't': web_token ← ttex_string;
  '=': web_token ← tverbatim;
  '\': web_token ← tforce_line;
  '!': web_token ← tunderline;
  '?': web_token ← tno_underline;
  ',': web_token ← tthin_space;
  '/': web_token ← tline_break;
  '|': web_token ← topt_line_break;
  '#': web_token ← tbig_line_break;
  '+': web_token ← tno_line_break;
  ';': web_token ← tpseudo_semi;
  othercases begin web_token ← tnul;
    end;
  end;
  end
```

This code is used in section 60.

66. This function converts a lower case alphabetic character to upper case.

```
⟨Function ch_upper return char 66⟩ ≡
function ch_upper(x : char): char;
  begin if x ∈ ['a' .. 'z'] then ch_upper ← chr(ord(x) − (ord('a') − ord('A')))
  else ch_upper ← x;
  end
```

This code is used in section 60.

67. Parsing. In the lexical analysis phase, all predefined Pascal tokens which are defined as operators are counted — except for user-defined program, sub-program, and macro names, and counting of *tvar* in sub-program parameters. They will be handled here in the parsing phase. We will also handle all items defined as an Halstead operand. That is, a token determined to be either a *tident*, *tnum*, *tstring*, or *tpreproc*.

```
⟨Function parser return next 67⟩ ≡
function parser: integer;
  var i, j: integer; id2: alpha; found: boolean;
  begin ⟨Set appropriate flag conditions 68⟩
  if islabel then   { don't count anything until islabel = false }
    begin if ¬(sym ∈ [tident, tnum, tstring, tpreproc, tnul]) then decr(count[sym])
    end
  else if (prev_sym ∈ [tprogram, tfunction, tprocedure]) ∧ (sym = tident) then
      begin ⟨Save user-defined name in user 74⟩;
      ⟨If id in opd, copy over count 75⟩;
      end
    else if isparam ∧ (sym = tvar) then
        begin incr(count[tvar]);
        debug outsym(tvar);
        gubed
        end
      else if (sym = tident) then
          begin ⟨Search for id in user, set found 76⟩;
          if found then subpgm ← i
          else begin subpgm ← 0; ⟨Count operand in opd,opdcnt 71⟩;
            end;
          end
        else if (sym = tbecomes) ∧ (prev_sym = tident) ∧ (subpgm > 0) then
            begin   { assignment to function or macro name, count as operand to tbecomes }
            decr(usercnt[subpgm]);
            debug write('␣*', user[subpgm], '␣', usercnt[subpgm] : 1, '*␣');
            gubedsubpgm ← 0; ⟨Count operand in opd,opdcnt 71⟩;
            end
          else if (sym ∈ [tstring, tpreproc]) then ⟨Count operand in opd,opdcnt 71⟩
            else if (prev_sym ∈ [ttimes, tslash, tlparen, teql, tcomma, tlt, tgt, tlbrack, tbecomes, tdotdot,
                    tneq, tleq, tgeq, tof, tcolon, tone_eql]) ∧ (sym ∈ [tplus, tminus]) then
                    { constant or simple type }
                sign ← true
              else if (sym = tnum) then
                  begin sign ← false; ⟨Copy id to id2, num to id 69⟩;
                  ⟨Count operand in opd,opdcnt 71⟩;
                  ⟨Copy id2 back to id 70⟩;
                  end
                else if (sym = telse) then   { don't count as an if stmt }
                    decr(count[tif]);   { adjust vg }
  ⟨Update cyclomatic complexity 15⟩;
  if (sym ≠ tnul) ∧ ¬(sym ∈ [tassumpt .. tz]) then prev_sym ← sym;
  if (sym ∈ [targument, tone_eql]) then prev_sym ← sym;
  parser ← code_state;
  end
```

This code is used in section 26.

68. ⟨Set appropriate flag conditions 68⟩ ≡

```
if prev_sym = tnul then prev_sym ← sym;   { first valid }
if sym = tlparen then
  begin isparam ← true; incr(parenbal);
  end
else if sym = trparen then
    begin isparam ← false; decr(parenbal);
    end
  else if sym = tlabel then islabel ← true;
if islabel ∧ (prev_sym = tsemi) then islabel ← false;
```

This code is used in section 67.

69. Set *id2* ← *id* and *id* ← *num*, so can use the same code to place the number into the *opd* array.

⟨Copy *id* to *id2*, *num* to *id* 69⟩ ≡

```
for i ← 1 to len_alpha do
  begin id2[i] ← id[i]; id[i] ← num[i];
  end
```

This code is used in section 67.

70. Restore the value of the last identifier read.

⟨Copy *id2* back to *id* 70⟩ ≡

```
for i ← 1 to len_alpha do id[i] ← id2[i]
```

This code is used in section 67.

71. If the token is *tident*, *tstring*, *tpreproc*, or *tnum*, then it is an operand. We will insert the operand *id* into the table *opd* if not already there, and increment the count in *opdcnt*. The contents of *opd* will be in the order that the operands are recognized in the input program. Thus searching will be sequential. Variable *numopd* (Halstead's η_2) will hold the number of elements in the array *opd*. The sum of all the counts of *opdcnt* is the total number of operands in the program (Halstead's N_2).

⟨Count operand in *opd*,*opdcnt* 71⟩ ≡

```
begin ⟨Search for id in opd, set found 72⟩;
if ¬found then ⟨Add id to opd 73⟩;
end
```

This code is used in sections 67, 67, 67, and 67.

72. Sequentially search the *opd* table for *id*. If found, increment count in parallel array *opdcnt*.

⟨Search for *id* in *opd*, set *found* 72⟩ ≡

```
i ← 1; found ← false;
while (i ≤ numopd) ∧ (¬found) do
  begin if comp_opd(i) then
    begin found ← true; incr(opdcnt[i]);
    debug write('␣', id, '-', opdcnt[i] : 1, '␣');
    gubed
    end
  else incr(i);
  end
```

This code is used in section 71.

73. Add *id* to the end of *opd* if it is not full yet, and increment count in *opdcnt*.
Declare a macro to copy contents of *id* to *opd*[#].

```
define copy_opd(#) ≡
        for j ← 1 to len_alpha do opd[#, j] ← id[j]
```

```
⟨Add id to opd 73⟩ ≡
  if (numopd < maxopd) then
    begin incr(numopd); copy_opd(numopd); opdcnt[numopd] ← 1;
    debug write('␣', id, '-', opdcnt[numopd] : 1, '␣');
    gubed
    end
  else opdfull ← true
```

This code is used in section 71.

74. We must keep user-defined names in a separate table *user* because they are counted as operators. Increment the count in *usercnt* if *prev_sym* = *tprogram*, but not if it is *tprocedure* or *tfunction*.
Declare a macro to copy contents of *id* to *user*[#].

```
define copy_user(#) ≡
        for j ← 1 to len_alpha do user[#, j] ← id[j]
```

```
⟨Save user-defined name in user 74⟩ ≡
  begin if (numuser < maxuser) then
    begin incr(numuser); copy_user(numuser);
    if prev_sym = tprogram then usercnt[numuser] ← 1
    else usercnt[numuser] ← 0;
    debug write('␣', id, '-', usercnt[numuser] : 1, '␣');
    gubed
    end
  else userfull ← true;
  end
```

This code is used in sections 29 and 67.

75. If *id* is found in *opd* and the current token is *tprocedure* or *tfunction*, then the procedure or function must have been called before it was declared (not allowed for macros). We need to remove it from the *opd* array and copy the count over to *usercnt*.

```
⟨If id in opd, copy over count 75⟩ ≡
  i ← 1; found ← false;
  while (i ≤ numopd) ∧ (¬found) do
    begin if comp_opd(i) then
      begin found ← true; usercnt[numuser] ← usercnt[numuser] + opdcnt[i]; opdcnt[i] ← 0;
      end
    else incr(i);
    end
```

This code is used in section 67.

76. Before we count *tident* token as an operand, we must first make sure it is not an operator (user-defined subprogram or macro name). Set $found = true$ and increment *usercnt* if *id* is found in *user*.

```
⟨Search for id in user, set found 76⟩ ≡
  begin i ← 1; found ← false;
  while (i ≤ numuser) ∧ (¬found) do
    begin if comp_user(i) then
      begin found ← true; incr(usercnt[i]);
      debug write('␣', id, '-', usercnt[i] : 1, '␣');
      gubed
      end
    else incr(i);
    end;
  end
```

This code is used in section 67.

77. Utilities for Parser. Before defining the utilities functions of the parser, let's define and intialize the various global variables we are using.

```
⟨Utilities of parser 77⟩ ≡
  ⟨Function comp_opd return equal 82⟩;
  ⟨Function comp_user return equal 83⟩;
```
This code is used in section 26.

78.
```
⟨Global types of the program 19⟩ +≡
  opdta = array [1 .. maxopd] of alpha;
  opdti = array [1 .. maxopd] of integer;
  userta = array [1 .. maxuser] of alpha;
  userti = array [1 .. maxuser] of integer;
```

79.
```
⟨Global constants of the program 21⟩ +≡
  maxopd = 600;  { maximum number of operands that can be handled }
  maxuser = 150;  { maximum number of user-defined subprograms and macros }
```

80.
```
⟨Global variables of the program 13⟩ +≡
user: userta;  { array of user-defined subprogram names }
usercnt: userti;  { array of counts corresponding to user }
numuser: integer;  { current number of user-defined subprograms }
userfull: boolean;  { tried to add too many elements to user }
opd: opdta;  { array of operands }
opdcnt: opdti;  { array of counts corresponding to opd }
numopd: integer;  { current number of operands }
opdfull: boolean;  { tried to add to many elements to opd }
prev_sym: token;  { previous sym recognized }
isparam: boolean;  { in parameter list }
parenbal: integer;  { add one if tlparen, subtract one if trparen }
islabel: boolean;  { in a label stmt }
sign: boolean;  { the tplus or tminus is a sign, not an operator }
subpgm: integer;  { save position just accessed in user }
```

81. Now, let's initialize those variables!

```
⟨Initialize global variables 64⟩ +≡
  numuser ← 0; eta2 ← 0; userfull ← false; opdfull ← false; prev_sym ← tnul; isparam ← false;
  subpgm ← 0; sign ← false; islabel ← false; numopd ← 0;
```

82. This function takes as a parameter the element number of array *opd* to be compared with *id*. The function will return *true* if they are equal.

```
⟨Function comp_opd return equal 82⟩ ≡
function comp_opd(i : integer): boolean;
  var j: integer; stop: boolean;
  begin stop ← false; j ← 1;
  while (j ≤ len_alpha) ∧ ¬stop do
    if (opd[i, j] ≠ id[j]) then stop ← true
    else incr(j);
  if ¬stop then comp_opd ← true
  else comp_opd ← false;
  end
```
This code is used in section 77.

83. This function takes as a parameter the element number of array *user* to be compared with *id*. The function will return *true* if they are equal.

⟨Function comp_user return *equal* 83⟩ ≡
function *comp_user*(*i* : *integer*): *boolean*;
  var *j*: *integer*; *stop*: *boolean*;
  begin *stop* ← *false*; *j* ← 1;
  while $(j \le len\_alpha) \land \neg stop$ **do**
    if $(user[i, j] \ne id[j])$ **then** *stop* ← *true*
    else *incr*(*j*);
  if ¬*stop* **then** *comp_user* ← *true*
  else *comp_user* ← *false*;
  end

This code is used in section 77.

84. Input and Output.

⟨ Utilities for input and output 84 ⟩ ≡
  ⟨ Procedure get_line; set *end_of_input* 86 ⟩;
  ⟨ Procedure sym_string; return *alpha* 87 ⟩;
  ⟨ Procedure outsym; print token 88 ⟩;
  ⟨ Procedure outcounts; print to *met_file* 89 ⟩;

This code is used in section 2.

85. ⟨ Open input and output files 85 ⟩ ≡

```
for i ← 1 to len_name do
   begin in_file[i] ← '␣'; out_file[i] ← '␣';
   end;
write('INPUT␣:␣'); i ← 1;
while (i ≤ len_name) ∧ (¬eoln) do
   begin read(cc); in_file[i] ← cc; incr(i);
   end;
reset(web_file, in_file); writeln; readln; write('OUTPUT:␣'); i ← 1;
while (i ≤ len_name) ∧ (¬eoln) do
   begin read(cc); out_file[i] ← cc; incr(i);
   end;
rewrite(met_file, out_file); writeln;
```

This code is used in section 64.

86. Procedure get_line will read one line of input from *web_file*. Global variable *end_of_input* is set to *true* when *eof* = *true*. This procedure must be modified to accommodate CHange files.

```
⟨Procedure get_line; set end_of_input 86⟩ ≡
procedure get_line;
  var ch: char; i: integer;
  begin if eof(web_file) then
    begin debug writeln('*****end␣of␣file'); writeln;
    gubed end_of_input ← true;
    end
  else begin debug writeln;
    gubed
    for i ← 1 to len_line do buffer[i] ← '␣';
    lbuf ← 0; cbuf ← 1;   { skip blank lines }
    while eoln(web_file) ∧ ¬(eof(web_file)) do readln(web_file);
    if ¬eof(web_file) then
      begin debug write('_');
      gubed
      while ¬eoln(web_file) ∧ (lbuf ≤ len_line − 2) ∧ ¬(eof(web_file)) do
        begin incr(lbuf); read(web_file, ch);
        debug write(ch);
        gubed buffer[lbuf] ← ch;
        end;   { check for too long of a line }
      if (lbuf = len_line − 1) ∧ (¬eoln(web_file)) then
        begin writeln; write('***warning,␣input␣line␣may␣have␣been␣truncated');
        writeln(met_file, '***warning,␣input␣line␣may␣have␣been␣truncated');
        end;
      readln(web_file);   { advance to next line }
      end;
    debug write('*L=', lbuf : 1);
    gubed
    end;
  end
```

This code is used in section 84.

87. Procedure sym_string sets *temp* to a string depending on what the *sym* token is. This is for printing purposes. The tokens commented out with meta-comments will not appear in the `TANGLE`d code. They are declared as tokens because they need to be recognized, but commented out because they are not to be counted as individual Pascal operators (they are either syntactic devices, or the second half of an operator pair).

⟨Procedure sym_string; return *alpha* 87⟩ ≡

```
procedure sym_string(sym : token; var temp : alpha);
  begin case sym of
  tnul: temp ← 'TNUL␣␣␣␣␣␣';
  tident: temp ← 'TIDENT␣␣␣␣';
  tnum: temp ← 'TNUM␣␣␣␣␣␣';
      { Pascal stuff }
  tand: temp ← 'and␣␣␣␣␣␣␣';
  tarray: temp ← 'array␣of␣␣';
  tbegin: temp ← 'begin␣end␣';
  tboolean: temp ← 'boolean␣␣␣';
  tcase: temp ← 'case␣end␣␣';
  tchar: temp ← 'char␣␣␣␣␣␣';
  @{tconst: temp ← 'tconst␣␣␣␣';
  @}tdiv: temp ← 'div␣␣␣␣␣␣␣';
  @{tdo: temp ← 'tdo␣␣␣␣␣␣␣';
  @}tdownto: temp ← 'downto␣␣␣␣';
  telse: temp ← 'if␣then␣el';
  @{tend: temp ← 'end␣␣␣␣␣␣␣';
  @}teof: temp ← 'eof␣␣␣␣␣␣␣';
  teoln: temp ← 'eoln␣␣␣␣␣␣';
  texternal: temp ← 'external␣␣';
  tfile: temp ← 'file␣␣␣␣␣␣';
  tfor: temp ← 'for␣do␣␣␣␣';
  tforward: temp ← 'forward␣␣␣';
  tfunction: temp ← 'function␣␣';
  tgoto: temp ← 'goto␣␣␣␣␣␣';
  tif: temp ← 'if␣then␣␣␣';
  tin: temp ← 'in␣␣␣␣␣␣␣␣';
  tinteger: temp ← 'integer␣␣␣';
  tlabel: temp ← 'label␣␣␣␣␣';
  tmod: temp ← 'mod␣␣␣␣␣␣␣';
  tnot: temp ← 'not␣␣␣␣␣␣␣';
  @{tof: temp ← 'tof␣␣␣␣␣␣␣';
  @}tor: temp ← 'or␣␣␣␣␣␣␣␣';
  totherwise: temp ← 'otherwise␣';
  tpacked: temp ← 'packed␣␣␣␣';
  tprocedure: temp ← 'procedure␣';
  tprogram: temp ← 'program␣␣␣';
  tread: temp ← 'read␣␣␣␣␣␣';
  treadln: temp ← 'readln␣␣␣␣';
  treal: temp ← 'real␣␣␣␣␣␣';
  trecord: temp ← 'record␣end';
  trepeat: temp ← 'repeat␣unt';
  tset: temp ← 'set␣␣␣␣␣␣␣';
  ttext: temp ← 'text␣␣␣␣␣␣';
  @{tthen: temp ← 'tthen␣␣␣␣␣';
```

```
@}tto: temp ← 'to␣␣␣␣␣␣␣␣';
@{ttype: temp ← 'ttype␣␣␣␣␣';
@}@{tuntil: temp ← 'tuntil␣␣␣␣';
@}tvar: temp ← 'var␣␣␣␣␣␣␣';
twhile: temp ← 'while␣do␣␣';
twith: temp ← 'with␣do␣␣␣';
twrite: temp ← 'write␣␣␣␣␣';
twriteln: temp ← 'writeln␣␣␣';
    { operators and delimeters }
tplus: temp ← '+␣␣␣␣␣␣␣␣␣';
tminus: temp ← '-␣␣␣␣␣␣␣␣␣';
ttimes: temp ← '*␣␣␣␣␣␣␣␣␣';
tslash: temp ← '/␣␣␣␣␣␣␣␣␣';
tlparen: temp ← '(␣)␣␣␣␣␣␣␣';
@{trparen: temp ← '␣␣␣␣␣␣␣␣␣␣';
@}teql: temp ← '=␣␣␣␣␣␣␣␣␣';
tcomma: temp ← ',␣␣␣␣␣␣␣␣␣';
tperiod: temp ← '.␣␣␣␣␣␣␣␣␣';
tlt: temp ← '<␣␣␣␣␣␣␣␣␣';
tgt: temp ← '>␣␣␣␣␣␣␣␣␣';
tsemi: temp ← ';␣␣␣␣␣␣␣␣␣';
tlbrack: temp ← '[␣]␣␣␣␣␣␣␣';
@{trbrack: temp ← '␣␣␣␣␣␣␣␣␣␣';
@}tbecomes: temp ← ':=␣␣␣␣␣␣␣␣';
tdotdot: temp ← '..␣␣␣␣␣␣␣␣';
tneq: temp ← '<>␣␣␣␣␣␣␣␣';
tleq: temp ← '<=␣␣␣␣␣␣␣␣';
tgeq: temp ← '>=␣␣␣␣␣␣␣␣';
tcaret: temp ← '^␣␣␣␣␣␣␣␣␣';
tcolon: temp ← ':␣␣␣␣␣␣␣␣␣';
    { special web and pascal stuff }
tassumpt: temp ← '|␣|␣␣␣␣␣␣␣';
tstring: temp ← '''...''␣␣␣';
tpreproc: temp ← '"..."␣␣␣␣␣';
tcomment: temp ← '{␣}␣␣␣␣␣␣␣';
targument: temp ← '#␣␣␣␣␣␣␣␣␣';
tdbl_eql: temp ← '==␣␣␣␣␣␣␣␣';
tone_eql: temp ← '=␣␣␣␣␣␣␣␣␣';
tis: temp ← '@<␣@>=␣␣';
tcall: temp ← '@<␣@>*;␣';
tabbrev: temp ← '...␣␣␣␣␣␣␣';
    { web commands }
tat: temp ← '@␣␣␣␣␣␣␣␣␣';
tnew_mod: temp ← '@b␣␣␣␣␣␣␣␣';
tstar_mod: temp ← '@*␣␣␣␣␣␣␣␣';
tdef: temp ← '@d␣␣␣␣␣␣␣␣';
tformat: temp ← '@f␣␣␣␣␣␣␣␣';
tbegin_code: temp ← '@p␣␣␣␣␣␣␣␣';
tmod_name: temp ← '@<␣␣@>␣␣';
@{tend_web: temp ← 'tend_web␣␣';
@}toctal: temp ← '@''␣␣␣␣␣␣';
thex: temp ← '@"␣␣␣␣␣␣␣';
```

```
  tcheck_sum: temp ← '@$␣␣␣␣␣␣␣';
  tbegin_comment: temp ← '@{␣␣@}␣␣';
  @{tend_comment: temp ← 'tend_comme';
  @}tjoin: temp ← '@&␣␣␣␣␣␣␣';
  troman: temp ← '@^␣␣@>␣␣';
  ttypewriter: temp ← '@.␣␣@>␣␣';
  tuser_def: temp ← '@:␣␣@>␣␣';
  ttex_string: temp ← '@t␣␣@>␣␣';
  tverbatim: temp ← '@=␣␣@>␣␣';
  tforce_line: temp ← '@\␣␣␣␣␣␣␣';
  tunderline: temp ← '@!␣␣␣␣␣␣␣';
  tno_underline: temp ← '@?␣␣␣␣␣␣␣';
  tthin_space: temp ← '@,␣␣␣␣␣␣␣';
  tline_break: temp ← '@/␣␣␣␣␣␣␣';
  topt_line_break: temp ← '@|␣␣␣␣␣␣␣';
  tbig_line_break: temp ← '@#␣␣␣␣␣␣␣';
  tno_line_break: temp ← '@+␣␣␣␣␣␣␣';
  tpseudo_semi: temp ← '@;␣␣␣␣␣␣␣';
  tx: temp ← '@x␣␣␣␣␣␣␣';
  ty: temp ← '@y␣␣␣␣␣␣␣';
  tz: temp ← '@z␣␣␣␣␣␣␣';
  othercases temp ← '␣␣␣␣␣␣␣␣␣␣';
  end;
  end
```

This code is used in section 84.

88. Procedure outsym prints one token to the screen, along with the number of occurrences.

⟨Procedure outsym; print token 88⟩ ≡

```
procedure outsym(sym : token);
  var temp: alpha; i: integer;
  begin sym_string(sym, temp);
  for i ← 1 to len_alpha − 1 do
    if (temp[i] ≠ '␣') ∨ (temp[i + 1] ≠ '␣') then write(temp[i]);
  if (temp[len_alpha] ≠ '␣') then write(temp[len_alpha]);
  write('-', count[sym] : 1); write('␣');
  end
```

This code is used in section 84.

89. This is a procedure to output counts to *met_file*.

⟨Procedure outcounts; print to *met_file* 89⟩ ≡

```
procedure outcounts;
  var i: token; s: alpha; j, k: integer;
  begin ⟨Output header to met_file 90⟩
  ⟨Output operators to met_file 91⟩
  ⟨Output operands to met_file 92⟩
  ⟨Output summary to met_file 93⟩
  ⟨Output web counts to met_file 94⟩
  ⟨Output warnings 95⟩
  end
```

This code is used in section 84.

90. ⟨ Output header to *met_file* 90 ⟩ ≡

```
writeln(met_file, 'for␣INPUT␣file␣:␣', in_file); writeln(met_file);
stat writeln(in_file, '␣');
tats
for i ← tident to tnum do
  begin sym_string(i, s); writeln(met_file, s, '␣:␣', count[i] : 4);
  stat write(count[i] : 4, '␣');
  tats
  end;
```

This code is used in section 89.

91. ⟨ Output operators to *met_file* 91 ⟩ ≡

```
writeln(met_file); writeln(met_file, 'OPERATORS'); writeln(met_file);
for i ← tand to tcolon do
  begin sym_string(i, s);
  if s ≠ '␣␣␣␣␣␣␣␣␣␣' then
    if count[i] ≠ 0 then writeln(met_file, s, '␣:␣', count[i] : 4);
  end;
for j ← 1 to numuser do
  begin for k ← 1 to len_alpha do write(met_file, user[j, k]);
  writeln(met_file, '␣:␣', usercnt[j] : 4);
  end;
writeln(met_file); writeln(met_file, 'eta␣1␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣', eta1 : 5);
writeln(met_file, 'total␣number␣of␣operators␣:␣', n1 : 5); writeln(met_file);
```

This code is used in section 89.

92. ⟨ Output operands to *met_file* 92 ⟩ ≡

```
writeln(met_file); writeln(met_file, 'OPERANDS'); writeln(met_file);
for j ← 1 to numopd do
  if (opdcnt[j] ≠ 0) then
    begin for k ← 1 to len_alpha do write(met_file, opd[j, k]);
    writeln(met_file, '␣:␣', opdcnt[j] : 4);
    end;
writeln(met_file); writeln(met_file, 'eta␣2␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣', eta2 : 5);
writeln(met_file, 'total␣number␣of␣operands␣␣:␣', n2 : 5); writeln(met_file);
```

This code is used in section 89.

93. ⟨Output summary to *met_file* 93⟩ ≡

```
writeln(met_file, '␣SUMMARY␣'); writeln(met_file);
writeln(met_file, 'numbered␣code␣sections␣', count[tnew_mod] + count[tstar_mod] : 5);
writeln(met_file, 'number␣of␣procedures␣␣␣', count[tprocedure] : 5);
writeln(met_file, 'number␣of␣functions␣␣␣␣', count[tfunction] : 5); writeln(met_file);
writeln(met_file, 'lines␣␣of␣limbo␣␣', lol : 3); writeln(met_file, 'lines␣␣of␣doc␣␣␣␣', lod : 3);
writeln(met_file, '␣␣per␣code␣sect.␣', lod/(count[tnew_mod] + count[tstar_mod]) : 5 : 2);
writeln(met_file, 'number␣of␣macros␣', lom : 3); writeln(met_file, 'lines␣␣of␣code␣␣␣', loc : 3);
writeln(met_file, '␣␣per␣code␣sect.␣', loc/count[tis] : 5 : 2); writeln(met_file);
writeln(met_file, 'VG␣␣␣␣␣␣␣:␣', vg : 8 : 2); writeln(met_file);
writeln(met_file, 'eta1␣␣␣␣␣:␣', eta1 : 8 : 2); writeln(met_file, 'eta2␣␣␣␣␣:␣', eta2 : 8 : 2);
writeln(met_file, 'n1␣␣␣␣␣␣␣:␣', n1 : 8 : 2); writeln(met_file, 'n2␣␣␣␣␣␣␣:␣', n2 : 8 : 2);
writeln(met_file);
stat writeln(count[tnew_mod] + count[tstar_mod] : 3, '␣', count[tprocedure] : 3, '␣', count[tfunction] : 3);
writeln(lol : 2, '␣', lod : 3, '␣', lod/(count[tnew_mod] + count[tstar_mod]) : 5 : 2, lom : 3, '␣', loc : 4, '␣',
    loc/count[tis] : 5 : 2); writeln(vg : 3, '␣', eta1 : 2, '␣', eta2 : 3, '␣', n1 : 4, '␣', n2 : 4, '␣');
tatswriteln(met_file, 'length␣␣␣:␣', hlength : 8 : 2); writeln(met_file, 'volume␣␣␣:␣', hvolume : 8 : 2);
writeln(met_file, 'effort␣␣␣:␣', heffort : 8 : 2);
writeln(met_file, 'time␣␣␣␣␣:␣', htime : 8 : 2, '␣seconds');
writeln(met_file, '␣␣␣␣␣␣␣␣␣:␣', (htime/60) : 8 : 2, '␣minutes');
writeln(met_file, '␣␣␣␣␣␣␣␣␣:␣', (htime/60)/60 : 8 : 2, '␣hours'); writeln(met_file);
stat write(hlength : 7 : 2, '␣', hvolume : 8 : 2, '␣', heffort : 11 : 2, '␣');
writeln(htime : 9 : 2, '␣', (htime/60) : 8 : 2, '␣', (htime/60)/60 : 6 : 2, '␣');
tats
```

This code is used in section 89.

94. ⟨Output web counts to *met_file* 94⟩ ≡

```
writeln(met_file); writeln(met_file, 'WEB␣Command␣Counts'); writeln(met_file);
for i ← tassumpt to tz do
  begin sym_string(i, s);
  if s ≠ '␣␣␣␣␣␣␣␣␣␣' then
    begin stat if (i ≠ tabbrev) then
      begin if (count[i] ≠ 0) then write(count[i] : 3, '␣')
      else write('␣␣.␣');
      end;
    if (i ∈ [tdef, tverbatim, tz]) then writeln;
    tats
    if count[i] ≠ 0 then writeln(met_file, s, '␣:␣', count[i] : 4);
    end;
  end;
```

This code is used in section 89.

95. These messages are output to warn the user that some calculations may be off because the *opd* or *user* overflowed.

```
⟨Output warnings 95⟩ ≡
  if (opdfull ∨ userfull) then
    begin writeln('***warning,␣metrics␣calculations␣may␣be␣incorrect␣');
    writeln(met_file, '***warning,␣metrics␣calculations␣may␣be␣incorrect␣'); writeln;
    writeln(met_file);
    if (opdfull) then
      begin writeln('␣␣␣␣␣␣␣␣␣␣␣number␣of␣operands␣may␣be␣inaccurate␣');
      writeln('␣␣␣␣␣␣␣␣␣␣␣numopd␣>=␣maxopd␣');
      writeln(met_file, '␣␣␣␣␣␣␣␣␣␣␣number␣of␣operands␣may␣be␣inaccurate');
      writeln(met_file, '␣␣␣␣␣␣␣␣␣␣␣numopd␣>=␣maxopd␣'); writeln; writeln(met_file);
      end;
    if (userfull) then
      begin writeln('␣␣␣␣␣␣␣␣␣␣␣number␣of␣operators␣may␣be␣inaccurate');
      writeln('␣␣␣␣␣␣␣␣␣␣␣numuser␣>=␣maxuser␣');
      writeln(met_file, '␣␣␣␣␣␣␣␣␣␣␣number␣of␣operators␣may␣be␣inaccurate');
      writeln(met_file, '␣␣␣␣␣␣␣␣␣␣␣numuser␣>=␣maxuser␣'); writeln; writeln(met_file);
      end;
    end;
```

This code is used in section 89.

96. Index.

APPENDIX C

A SAMPLE LITERATE PROGRAM and its OUTPUT

The Knight's Tour

1. Introduction. The following program is based on the "Knight's Tour" algorithm found on pages 137–142 of Niklaus Wirth's *Algorithms + Data Structures = Programs* (pages 148–152 in the 1986 edition, renamed *Algorithms and Data Structures*), translated into the WEB language.

2. This program has no input because we want to keep it rather simple. The result of the program will be the solution to the problem, which will be written to the *output* file.

In true top-down tradition, we lay out the entire program as a skeleton which will be filled in later.

program *knights_tour*(*output*);
  const ⟨Constants of the program 6⟩
  type ⟨Types of the program 7⟩
  var ⟨Variables of the program 8⟩
    ⟨Recursive procedure definitions 12⟩
    begin ⟨Initialize the data structures 19⟩;
    ⟨Perform the Knight's Tour and print the results 21⟩;
    end.

3. Here are some macros for common programming idioms.

define *incr*(#) $\equiv$ # $\leftarrow$ # + 1 { increase a variable by unity }
define *decr*(#) $\equiv$ # $\leftarrow$ # − 1 { decrease a variable by unity }

4. We shall proceed to build the program in pieces, following the text of Wirth's book and describing the structures and algorithms in more or less the same order in which he describes them. Part of the time we will be designing top-down; at other times, bottom-up; but always in the order that contributes more to the understanding of the program. One difference between this description and Wirth's is that we will use meaningful variable names (rather than names such as u, v, a, b, c). We can get away with this because we don't have to worry about the entire program's being listed in one place in narrow columns. It is broken up into small pieces and spread over several pages.

5. The Tour. For those not familiar with the Knight's Tour, it is a classic computer science problem involving a knight, moving according to the rules of chess, which attempts to move to every square of a chessboard once and only once. To implement it, we use what is known as a "backtracking" algorithm, which is a trial-and-error search for a solution, sometimes referred to as a "brute-force" approach. We start at a beginning position and try every path leading from that position (there are 8 possible knight moves from a given position) and then every path from each of those positions, etc. We folllow every path until either a complete solution is found or the first failure occurs (the square is not on the board or has already been visited). If we get a failure, we "backtrack" to the previous good move and start again from there.

6. The board is a $max \times max$ square and the number of squares is max^2. Since this program has no input, we declare the value $max = 5$ as a compile-time constant.

⟨Constants of the program 6⟩ ≡
  max = 5; *number_of_squares* = 25;

This code is used in section 2.

7. The obvious way to define the board is as an array of two dimensions where the indexes range from 1 to *max*.

⟨Types of the program 7⟩ ≡
  index = 1 .. *max*;

This code is used in section 2.

8. Boolean values for the squares would be sufficient if we only wanted to know which squares had been visited, but we also wish to know the *order* of the visits, so we define *board* as a two-dimensional array of ordinal values ranging from 0 to *number_of_squares*. If the value at *board*[*row*, *col*] = 0 then the square at that position has not been visited and is a candidate for a visit. Otherwise *board*[*row*, *col*] = *i*, which indicates that the square was visited on the *i*th move. The total number of moves (including the first one) = *number_of_squares*.

  define *empty* = 0

⟨Variables of the program 8⟩ ≡
board: **array** [*index*, *index*] **of** *empty* .. *number_of_squares*;

See also sections 18 and 27.

This code is used in section 2.

9. Here we initialize all positions on the board to *empty*. This code is placed in a program scrap separate from all of the other initialization code for reasons explained later.

⟨Initialize the Knight's Tour board 9⟩ ≡
  for *i* ← 1 **to** *max* **do**
    for *j* ← 1 **to** *max* **do** *board*[*i*, *j*] ← *empty*;

This code is used in sections 21, 22, and 23.

10. Two local variables, *targ_row* and *targ_col*, are the coordinates of *target_square*, the one to which we wish to move next. Before we attempt the move, we must first ensure that *target_square* is on the board ($1 \le targ\_row \le max$ and $1 \le targ\_col \le max$). Then we must determine whether it is available for knight placement (*target_square* = *empty*). We provide some macros for manipulation of *target_square*.

Side note: in Wirth's book, the *target_position_valid* test is later changed to be a check for inclusion in the set [1 .. *max*] rather than discrete comparisons to 1 and *max*. While this is more efficient, it does not necessarily aid understanding of the algorithm, so we will not bother.

define *valid_row_or_column*(#) $\equiv ((1 \le \#) \land (\# \le max))$
define *target_position_valid* $\equiv$ (*valid_row_or_column*(*targ_row*) $\land$ *valid_row_or_column*(*targ_col*))
define *target_square* $\equiv$ *board*[*targ_row*, *targ_col*]
define *target_empty* $\equiv$ (*target_square* = *empty*)
define *set_square_to*(#) $\equiv$ *target_square* $\leftarrow$ #

⟨Local variables of the Knight's Tour procedure 10⟩ $\equiv$
targ_row, *targ_col*: *integer*; {row and column of target square}

See also section 11.

This code is used in section 12.

11. There are 8 possible knight moves from any given position. Since we normally are going to attempt all 8, we will define *candidate_move* to hold the move index.

define *number_of_legal_knight_moves* = 8

⟨Local variables of the Knight's Tour procedure 10⟩ $+\equiv$
candidate_move: 0 .. *number_of_legal_knight_moves*;

12. We are ready to define the procedure which actually does the search. It must be declared as a procedure rather than as simple inline code, since it is recursive. As the procedure is entered for a particular move number and current position, the moves possible from that position are attempted. The Boolean result *successful* is set if one of the 8 paths results in a solution. The procedure passes the value of *successful* back to the calling procedure, which will pass it to its own caller, etc., all the way back up to the original call.

⟨Recursive procedure definitions 12⟩ $\equiv$
procedure *try_knight_move*(*move_number* : *integer*; *row*, *col* : *index*; **var** *successful* : *Boolean*);
  var ⟨Local variables of the Knight's Tour procedure 10⟩
    begin ⟨Try the moves possible from the current position until solution is found or all have been tried 13⟩;
    end;

This code is used in section 2.

13. Each of the 8 possible moves is attempted. If the move can be made, it is recorded and a move from the new position is attempted. If *successful* is ever *true*, it can only mean that a complete solution has been found and the search is terminated. If *successful* is false, the remaining candidate moves are tried.

define *no_more_candidates* $\equiv$ (*candidate_move* = *number_of_legal_knight_moves*)

⟨Try the moves possible from the current position until solution is found or all have been tried 13⟩ $\equiv$
  candidate_move $\leftarrow$ 0;
  repeat *incr*(*candidate_move*); *successful* $\leftarrow$ *false*;
    ⟨Set the coordinates of the next move as defined by the rules of chess 17⟩;
    ⟨Record the move if acceptable and try to make further moves; set *successful* if solution is found 14⟩;
  until *successful* $\lor$ *no_more_candidates*;

This code is used in section 12.

14. The condition *move_is_acceptable* is equivalent to the conditions (*target_position_valid* $\land$ *target_empty*). Because of the realities of computer memory addressing, if we find that the condition *target_position_valid* is not *true*, we cannot perform the second test because the array indexes *targ_row* or *targ_col* are not valid and the contents of *target_square* cannot be accessed. We have to test these two conditions with separate nested **if** statements ("**if** *target_position_valid* **then if** *target_empty* **then**"). In the future, when the ANSI Extended Pascal Standard is adopted, its short-circuiting **and_then** operator will make this unnecessary, but we have to handle it manually in the meantime.

Once we have determined that the move is acceptable, we record it. If *board_not_full* is true, we try the next knight move. If *board_not_full* if false (i.e., the board *is* full), it means we have found a solution to the problem, so we set *successful* to true, which will terminate the tour.

define *board_not_full* $\equiv$ *move_number* $<$ *number_of_squares*
define *record_move* $\equiv$ *set_square_to*(*move_number*)

⟨Record the move if acceptable and try to make further moves; set *successful* if solution is found 14⟩ $\equiv$
if *target_position_valid* **then**
  if *target_empty* **then**
    begin *record_move*;
    if *board_not_full* **then** ⟨Try further knight moves and erase this move if not successful 15⟩
    else *successful* $\leftarrow$ *true*;
    end;

This code is used in section 13.

15. Here we try the next move by having the procedure call itself recursively, with the *move_number* incremented by one and *targ_row*, *targ_col* as the position of the move. If a failure occurs on that move or any move that follows it (*successful* = *false*), we erase the move we just made (this is the "backtracking" part) and continue looking.

Important note: the **begin** and **end** statements are *critical* for this particular section, since they are used as a **then** clause in the previous section and the program text is simply inserted verbatim. Without the **begin** and **end** , the call to the *try_knight_move* procedure alone becomes the **then** clause, which will cause a syntax error when the dangling **else** clause is processed.

define *next_move* $\equiv$ *move_number* $+ 1$
define *erase_move* $\equiv$ *set_square_to*(*empty*)

⟨Try further knight moves and erase this move if not successful 15⟩ $\equiv$
begin *try_knight_move*(*next_move*, *targ_row*, *targ_col*, *successful*);
if $\neg$*successful* **then** *erase_move*;
end

This code is used in section 14.

16. We now consider the moves a knight is allowed to make. From any given position there are 8 possible moves, not all of which are necessarily on the board. A knight makes a two-part L-shaped move, where the first part is either one or two squares in a nondiagonal direction, and the second part is one or two squares in a direction perpendicular to the first. The number of squares is never the same for the two parts; if the knight is moved one square during the first part, then it is moved two squares during the second, and vice versa.

	$\odot$	$\Longleftrightarrow$	$\odot$	
$\odot$		$\Uparrow$		$\odot$
$\Updownarrow$	$\Leftarrow$	$\spadesuit$	$\Rightarrow$	$\Updownarrow$
$\odot$		$\Downarrow$		$\odot$
	$\odot$	$\Longleftrightarrow$	$\odot$	

The '$\spadesuit$' represents a knight. Doesn't it sort of look like one? (You have to use some imagination. Okay, a *lot* of imagination.) The '$\odot$' characters represent the legal destinations from the current position.

17. Rather than go through a complicated algorithm, we simply initialize a pair of tables, *row_deltas* and *col_deltas*, containing values to be added to the current position to get the target position. For each of the 8 moves which are possible from the current position, *row_deltas*[*candidate_move*] is added to the current row to get *targ_row* and *col_deltas*[*candidate_move*] provides the same service for *targ_col*.

⟨ Set the coordinates of the next move as defined by the rules of chess 17 ⟩ ≡
targ_row ← *row* + *row_deltas*[*candidate_move*]; *targ_col* ← *col* + *col_deltas*[*candidate_move*];

This code is used in section 13.

18. If we are going to use these arrays, it might be helpful to define them to prevent the Pascal compiler from complaining bitterly.

⟨ Variables of the program 8 ⟩ +≡
row_deltas, *col_deltas*: **array** [1 .. *number_of_legal_knight_moves*] **of** −2 .. 2;

19. Even though the compiler is now happy, if we don't initialize the arrays with the proper delta values, we will take the knight on a route more like the Drunkard's Walk than the Knight's Tour.

⟨ Initialize the data structures 19 ⟩ ≡
row_deltas[1] ← 2; *col_deltas*[1] ← 1; *row_deltas*[2] ← 1; *col_deltas*[2] ← 2; *row_deltas*[3] ← −1;
col_deltas[3] ← 2; *row_deltas*[4] ← −2; *col_deltas*[4] ← 1; *row_deltas*[5] ← −2; *col_deltas*[5] ← −1;
row_deltas[6] ← −1; *col_deltas*[6] ← −2; *row_deltas*[7] ← 1; *col_deltas*[7] ← −2; *row_deltas*[8] ← 2;
col_deltas[8] ← −1;

This code is used in section 2.

20. Now it's time to start the tour. For starting position x_0y_0 we select (1,1). Since this is the first move, we set *board*[1, 1] = 1. We then call the *try_knight_move* procedure with the proper parameters for move 2 to set events in motion. After all of the moves have been completed, we print out the board. The result should be the same as the first part of Table 3.1 on page 141 (page 151, 1986 edition) of Wirth's book, which is reproduced here.

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

21. We define a macro to initialize the first move and start the tour. Note: the call to *try_knight_move* in the definition of *do_the_tour_starting_at* appears to have the wrong number of parameters. This procedure requires four parameters and we seem to be passing only three. However, since a macro parameter is really just a simple string substitution, we will really be replacing the # character with two parameters at once: the row and column of the starting position. Since the comma is included in the substitution, the expanded text will have the proper four parameters.

define *do_the_tour_starting_at*(#) ≡ *board*[#] ← 1; *try_knight_move*(2, #, *successful*);

⟨Perform the Knight's Tour and print the results 21⟩ ≡
⟨Initialize the Knight's Tour board 9⟩;
do_the_tour_starting_at(1, 1); ⟨Print the results of the Knight's Tour 26⟩;

See also sections 22 and 23.
This code is used in section 2.

22. Since Table 3.1 in the book shows a second solution obtained with a different first move (3,3),

23	10	15	4	25
16	5	24	9	14
11	22	1	18	3
6	17	20	13	8
21	12	7	2	19

we decide to run the tour again to duplicate that result as well. We decline the third test in Table 3.1, because it requires a different value of *max*, which we cannot change at run-time as the program is currently designed. Before rerunning the test, we must remember to reinitialize the board. This is the reason that the board initialization code was separated from all of the other initializations; it is executed more than once. We use the same name when defining the code for this section as for the previous section, which will cause the second test to follow immediately after the first in the Pascal program.

⟨Perform the Knight's Tour and print the results 21⟩ +≡
⟨Initialize the Knight's Tour board 9⟩;
do_the_tour_starting_at(3, 3); ⟨Print the results of the Knight's Tour 26⟩;

23. The 1986 edition shows yet another solution, with (2,4) as the starting move.

23	4	9	14	25
10	15	24	1	8
5	22	3	18	13
16	11	20	7	2
21	6	17	12	19

⟨Perform the Knight's Tour and print the results 21⟩ +≡
⟨Initialize the Knight's Tour board 9⟩;
do_the_tour_starting_at(2, 4); ⟨Print the results of the Knight's Tour 26⟩;

24. The output phase. The job of printing is not as interesting as the problem itself, but it must be done sooner or later, so we may as well get it over with.

25. In order to keep this program reasonably free of notations that are uniquely Pascalesque, a few macro definitions for low-level output instructions are introduced here. All of the output-oriented commands in the remainder of the program will be stated in terms of three simple primitives called *new_line*, *print_string*, and *print_integer*.

define *width* = 3 { width of an integer field }
define *print_string*(#) ≡ *write*(#); { put a given string into the *output* file }
define *print_integer*(#) ≡ *write*(# : *width*) { print an integer of *width* character positions }
define *new_line* ≡ *writeln* { advance to a new line in the *output* file }

26. ⟨ Print the results of the Knight's Tour 26 ⟩ ≡

```
if successful then
  for i ← 1 to max do
    begin for j ← 1 to max do print_integer(board[i, j]);
    new_line;
    end
else print_string('no␣solution');
new_line; new_line;
```

This code is used in sections 21, 22, and 23.

27. We define a few more odd variables.

⟨ Variables of the program 8 ⟩ +≡
i, *j*: *index*; { temporary index variables }
successful: *Boolean*; { The Knight's Tour was successful }

28. Index.

TANGLEd Version of *knights.web*

```
{2:}program knightstour(output);const{6:}max=5;numberofsquares=25;{:6}
type{7:}index=1..max;{:7}var{8:}
board:array[index,index]of 0..numberofsquares;{:8}{18:}
rowdeltas,coldeltas:array[1..8]of-2..2;{:18}{27:}i,j:index;
successful:Boolean;{:27}{12:}procedure tryknightmove(movenumber:integer;
row,col:index;var successful:Boolean);var{10:}targrow,targcol:integer;
{:10}{11:}candidatemove:0..8;{:11}begin{13:}candidatemove:=0;
repeat candidatemove:=candidatemove+1;successful:=false;{17:}
targrow:=row+rowdeltas[candidatemove];
targcol:=col+coldeltas[candidatemove];{:17};{14:}
if(((1<=targrow)and(targrow<=max))and((1<=targcol)and(targcol<=max)))
then if(board[targrow,targcol]=0)then begin board[targrow,targcol]:=
movenumber;if movenumber<numberofsquares then{15:}
begin tryknightmove(movenumber+1,targrow,targcol,successful);
if not successful then board[targrow,targcol]:=0;end{:15}
else successful:=true;end;{:14};until successful or(candidatemove=8);
{:13};end;{:12}begin{19:}rowdeltas[1]:=2;coldeltas[1]:=1;
rowdeltas[2]:=1;coldeltas[2]:=2;rowdeltas[3]:=-1;coldeltas[3]:=2;
rowdeltas[4]:=-2;coldeltas[4]:=1;rowdeltas[5]:=-2;coldeltas[5]:=-1;
rowdeltas[6]:=-1;coldeltas[6]:=-2;rowdeltas[7]:=1;coldeltas[7]:=-2;
rowdeltas[8]:=2;coldeltas[8]:=-1;{:19};{21:}{9:}
for i:=1 to max do for j:=1 to max do board[i,j]:=0;{:9};board[1,1]:=1;
tryknightmove(2,1,1,successful);;{26:}
if successful then for i:=1 to max do begin for j:=1 to max do write(
board[i,j]:3);writeln;end else write('no solution');;writeln;writeln;
{:26};{:21}{22:}{9:}for i:=1 to max do for j:=1 to max do board[i,j]:=0;
{:9};board[3,3]:=1;tryknightmove(2,3,3,successful);;{26:}
if successful then for i:=1 to max do begin for j:=1 to max do write(
board[i,j]:3);writeln;end else write('no solution');;writeln;writeln;
{:26};{:22}{23:}{9:}for i:=1 to max do for j:=1 to max do board[i,j]:=0;
{:9};board[2,4]:=1;tryknightmove(2,2,4,successful);;{26:}
if successful then for i:=1 to max do begin for j:=1 to max do write(
board[i,j]:3);writeln;end else write('no solution');;writeln;writeln;
{:26};{:23};end.{:2}
```

WEB Source Code: *knights.web*

```
%limbo material
%
% from p. 256 of Wayne Sewell's book
%
\def\WEB{{\tt WEB}}
\def\title{The Knight's Tour}
\countdef\pageno=108 \pageno=109
\def\9#1{}%this is used for the sort keys in the index via @@:sort key}{entry@@>
%
\def\smalline{height2pt&\omit&&&&&&&&&\cr}
\def\hrline{\multispan{11}\hrulefill\cr}
%
%
%
@* Introduction.
The following program is based on the ``Knight's Tour'' algorithm
@^Knight's Tour@>
found on pages 137--142 of Niklaus Wirth's {\sl Algorithms + Data
@^Wirth, Niklaus@>
Structures = Programs}
(pages 148--152 in the 1986 edition, renamed {\sl Algorithms and Data
Structures}),
translated into the \WEB\ language. @.WEB@>


@ This program has no input because we want to keep it rather simple.
The result of the program will be the solution to the
problem, which will be written to the |output| file.

In true top-down tradition, we lay out the entire program as
a skeleton which will be filled in later.


@p
program knights_tour(@!output);
const @<Constants of the program@>@;
type @<Types of the program@>@;
var @<Variables of the program@>@;@/
@<Recursive procedure definitions@>
begin @/
@<Initialize the data structures@>;
@<Perform the Knight's Tour and print the results@>;
end.


@ Here are some macros for common programming idioms.
```

```
@d incr(#) == #:=#+1 {increase a variable by unity}
@d decr(#) == #:=#-1 {decrease a variable by unity}
@ We shall proceed to build the program in pieces, following the text of
Wirth's book and describing the structures and algorithms in more or less the
same order in which he describes them.  Part of the time we will be designing
top-down; at other times, bottom-up; but always in the order that contributes
more to the understanding of the program.  One difference between this
description and Wirth's is that we will use meaningful variable names (rather
than names such as |u,v,a,b,c|).  We can get away with this
because we don't have to worry about the entire program's being listed in one
place in narrow columns.  It is broken up into small pieces and
spread over several pages.
@* The Tour.
@!@^Knight's Tour@>
For those not familiar with the Knight's Tour, it is a classic computer science
problem involving a knight, moving according to the rules of chess, which
attempts to move to every square of a chessboard once and
only once.
To implement it, we use what is known as a ``backtracking'' algorithm,
@!@^backtracking algorithm@>
which is a trial-and-error search for a solution, sometimes referred to as a
``brute-force'' approach.  We start at a beginning position and try every path
leading from that position (there are 8 possible knight moves from a given
position) and then every path from each of those positions, etc.  We folllow
every path until either a complete solution is found or the first failure occurs
(the square is not on the board or has already been visited).  If we
get a failure, we ``backtrack'' to the previous good move and start again from
there.

@ The board is a $max \times max$ square and the
number of squares is $|max|^2$.
Since this program has no input, we declare the value |max=5| as a compile-time
constant.
@<Constants of the program@>=
@!max = 5;
@!number_of_squares = 25;
@ The obvious way to define the board is
as an array of two dimensions where the indexes range from 1 to |max|.
@<Types of the program@>=
@!index=1..max ;

@ Boolean values for the squares would be sufficient if we only wanted to know
which squares had been visited, but we also wish to know the {\it order} of
the visits, so we define |board| as a two-dimensional array of
ordinal values ranging from 0 to |number_of_squares|.
If the value at |board[row,col]=0|
then the square at that position has not been visited and
is a candidate for a visit.  Otherwise |board[row,col]=i|, which indicates
that the square was visited on the |i|th move.  The total number of moves
(including the first one) |=number_of_squares|.
@d empty=0
@<Variables of the program@>=
@!board : array[index,index] of empty .. number_of_squares;
@ Here we initialize all positions on the board to |empty|.
```

```
This code is placed in a
program scrap separate from all of the other initialization code
for reasons explained later.
@<Initialize the Knight's Tour board@>=
for i := 1 to max do
  for j := 1 to max do
    board[i,j] := empty;
@ Two local variables, |targ_row| and |targ_col|, are the
coordinates of |target_square|, the one to which we wish to move next.
Before we attempt the move, we must first ensure that
|target_square| is on the board (|1<=targ_row<=max| and
|1<=targ_col<=max|).  Then we must determine whether it is available
for knight placement (|target_square=empty|).
We provide some macros for manipulation
of |target_square|.

Side note: in Wirth's book,
the |target_position_valid| test is later changed to be a check
for inclusion in the set |[1..max]| rather than discrete comparisons to 1 and
|max|.  While this is more efficient, it does not necessarily aid understanding
of the algorithm, so we will not bother.
@d valid_row_or_column(#)== @|
( (1 <= #) and (# <= max) )
@d target_position_valid== @|
(valid_row_or_column(targ_row) and @|
valid_row_or_column(targ_col) )
@d target_square==board[targ_row,targ_col]
@d target_empty== @| (target_square = empty)
@d set_square_to(#)== @| target_square := #
@<Local variables of the Knight's Tour procedure@>=
@!targ_row,@!targ_col : integer; {row and column of target square}
@ There are 8 possible knight moves from any given position.
Since we normally are going to attempt all 8, we will
define |candidate_move| to hold the move index.
@d number_of_legal_knight_moves=8
@<Local variables of the Knight's Tour procedure@>=
@!candidate_move : 0..number_of_legal_knight_moves ;

@ We are ready to define the procedure which actually does the search.
It must be declared as a procedure rather than as simple inline code, since it
is recursive.  As the procedure is entered for a particular move number and
current position, the moves possible from that position are attempted.
The Boolean result |successful| is set if one of the 8 paths results
in a solution.
The procedure passes the value of |successful| back to
the calling procedure, which will pass it to its own caller, etc., all the
way back up to the original call.
@<Recursive procedure definitions@>=
procedure try_knight_move(@!move_number:integer ;
                  @!row,@!col:index ;
                  var successful:Boolean) ;

 var @<Local variables of the Knight's Tour procedure@>
```

```
begin
@<Try the moves possible from the current position until solution is found
or all have been tried@>;
end;
@ Each of the 8 possible moves is attempted.
If the move can be made, it is recorded and a
move from the new position is attempted.
If |successful| is ever |true|,
it can only mean that a complete solution has
been found and the search is terminated.
If |successful| is false, the remaining
candidate moves are tried.


@d no_more_candidates== @|
(candidate_move = number_of_legal_knight_moves)


@<Try the moves...@>=
candidate_move := 0 ;
repeat
  incr(candidate_move) ;
  successful := false ;
  @<Set the coordinates of the next move as defined
         by the rules of chess@>;
  @<Record the move if acceptable and try to make
       further moves; set |successful| if
       solution is found@>;
until successful or no_more_candidates ;

@ The condition \\{move\_is\_acceptable} is equivalent to
the conditions (|target_position_valid and target_empty|).
Because of the realities of computer memory addressing, if
we find that the condition |target_position_valid|
is not |true|, we cannot perform the second test because
the array indexes |targ_row| or |targ_col| are not valid and
the contents of |target_square| cannot be accessed.
We have to test these two conditions with separate nested |if| statements
(``|if target_position_valid then if target_empty then|'').
In the future, when the ANSI Extended Pascal Standard is adopted, its
short-circuiting \&{and\_then} operator will make this unnecessary, but we have
to handle it manually in the meantime.

Once we have determined that the move is acceptable, we record it.  If
|board_not_full| is true, we try the next knight move.
If |board_not_full| if false (i.e., the board {\it is} full),
it means we have found a solution to the
problem, so we set |successful| to true, which will terminate the tour.
@d board_not_full==@|
move_number < number_of_squares
@d record_move== @| set_square_to(move_number)
@<Record the move if...@>=
if target_position_valid then
  if target_empty then
```

```
    begin
      record_move ;
      if board_not_full then @|
        @<Try further knight moves and erase this move if not successful@>
      else
        successful := true;

    end;
@ Here we try the next move by having the procedure call itself recursively,
with the |move_number| incremented by one and
|targ_row,targ_col| as the position of the move.
If a failure occurs on that move or any move that follows it
(|successful=false|), we erase the move we just made (this is the
``backtracking'' part) and continue looking.
@^backtracking algorithm@>

Important note: the |begin| and
|end| statements are {\it critical} for this particular section, since they
are used as a |then| clause in the previous section
and the program text is simply inserted verbatim.
Without the |begin| and |end|, the call to the |try_knight_move| procedure
alone becomes
the |then| clause, which will cause a syntax error when the dangling
|else| clause is processed.
@d next_move==move_number + 1
@d erase_move==set_square_to(empty)
@<Try further...@>=
begin
  try_knight_move(next_move,targ_row,targ_col,successful) ;
  if not successful then
      erase_move;
end
@ We now consider the moves
a knight is allowed to make.  From any given
position there are 8 possible moves, not all of which are necessarily on
the board.  A knight makes a two-part L-shaped move, where the first part
is either one or two squares in a nondiagonal direction, and the second
part is one or two squares in a direction perpendicular to the first.
The number of squares
is never the same for the two parts; if the knight is moved one square during
the first part, then it is moved two squares during the second, and vice versa.

$$\vbox{
\offinterlineskip
\halign{ \vrule # & \strut\  # \ & \vrule # & \  # \  &
      \vrule # & \  # & \vrule # & \  # & \vrule # & \  # & \vrule # \cr
\hrline\smalline
&&&  $\odot$ && $\Longleftrightarrow$   && $\odot$ &&& \cr
\smalline\hrline\smalline
&  $\odot$  &&&& $\Uparrow$   &&&& $\odot$ & \cr
\smalline\hrline\smalline
&  $\Updownarrow$  &&  $\Leftarrow$  && $\spadesuit$ &&
        $\Rightarrow$    && $\Updownarrow$   & \cr
\smalline\hrline\smalline
```

```
& $\odot$ &&&& $\Downarrow$  &&&& $\odot$ & \cr
\smalline\hrline\smalline
&&& $\odot$ &&  $\Longleftrightarrow$ && $\odot$ &&& \cr
\smalline\hrline
}}$$

The `$\spadesuit$' represents a knight.  Doesn't it sort
of look like one?  (You have to use some imagination.  Okay, a {\it lot}
of imagination.)  The `$\odot$' characters represent the legal
destinations from the current position.

@ Rather than go through a complicated algorithm, we simply
initialize a pair of tables, |row_deltas| and |col_deltas|, containing
values to be added to the current position to get the target position.
For each of the 8 moves which are possible from the current position,
|row_deltas[candidate_move]| is added to the current row to get |targ_row| and
|col_deltas[candidate_move]| provides the same service for |targ_col|.
@<Set the coordinates of the next move as defined by the rules of chess@>=
targ_row := row + row_deltas[candidate_move] ;
targ_col := col + col_deltas[candidate_move] ;

@ If we are going to use these arrays, it might be helpful to define them
to prevent the Pascal compiler from complaining bitterly.
@<Variables of the program@>=
@!row_deltas,@!col_deltas : @|
array @| [1..number_of_legal_knight_moves] @| of @| -2..2 ;
@ Even though the compiler is now happy, if we don't initialize the arrays with
the proper delta values, we will take the knight on a route more like the
Drunkard's Walk than the Knight's Tour.
@^Drunkard's Walk@>
@^Knight's Tour@>
@<Initialize the data structures@>=
    row_deltas[1] := 2 ;        col_deltas[1] := 1 ;@|
    row_deltas[2] := 1 ;        col_deltas[2] := 2 ;@|
    row_deltas[3] := - 1 ;      col_deltas[3] := 2 ;@|
    row_deltas[4] := - 2 ;      col_deltas[4] := 1 ;@|
    row_deltas[5] := - 2 ;      col_deltas[5] := - 1 ;@|
    row_deltas[6] := - 1 ;      col_deltas[6] := - 2 ;@|
    row_deltas[7] := 1 ;        col_deltas[7] := - 2 ;@|
    row_deltas[8] := 2 ;        col_deltas[8] := - 1 ;
@ Now it's time to start the tour.  For starting position $x_{0}y_{0}$
@^Knight's Tour@>
we select (1,1).  Since this is the first move, we set |board[1,1]=1|.  We
then call the |try_knight_move| procedure
with the proper parameters for move 2 to set events in
motion.  After all of the moves have been completed, we print out the board.
The result should be the same as the first part of Table 3.1 on page 141
(page 151, 1986 edition) of
Wirth's book, which is reproduced here.


$$\vbox{
\offinterlineskip
\halign{ \vrule # & \strut\  # \ & \vrule # & \  # \  &
```

```
      \vrule # & \  # & \vrule # & \  # & \vrule # & \  # & \vrule # \cr
\hrline\smalline
& 1   &&  6 && 15 && 10 && 21 & \cr
\smalline\hrline\smalline
& 14  &&  9 && 20 &&  5 && 16 & \cr
\smalline\hrline\smalline
& 19  &&  2 &&  7 && 22 && 11 & \cr
\smalline\hrline\smalline
& 8   && 13 && 24 && 17 &&  4 & \cr
\smalline\hrline\smalline
& 25  && 18 &&  3 && 12 && 23 & \cr
\smalline\hrline
}}$$

@ We define a macro to initialize the first move and start the tour.
Note: the call to |try_knight_move| in the definition
of |do_the_tour_starting_at| appears to have
the wrong number of parameters.  This procedure requires four parameters and
we seem to be passing only three.  However, since a macro parameter is really
just a simple string substitution, we will really be replacing the \#
character with two parameters at once: the row and column of the starting
position.  Since the comma is included in the substitution, the expanded text
will have the proper four parameters.
@d do_the_tour_starting_at(#)==board[#] := 1 ;try_knight_move(2,#,successful) ;
@<Perform the Knight's Tour and print the results@>=
@<Initialize the Knight's Tour board@>;
do_the_tour_starting_at(1,1) ;
@<Print the results of the Knight's Tour@>;

@ Since Table 3.1 in the book shows a second solution obtained with
a different first move (3,3),

$$\vbox{
\offinterlineskip
\halign{ \vrule # & \strut\  # \ & \vrule # & \  # \  &
      \vrule # & \  # & \vrule # & \  # & \vrule # & \  # & \vrule # \cr
\hrline\smalline
& 23  && 10 && 15 &&  4 && 25 & \cr
\smalline\hrline\smalline
& 16  &&  5 && 24 &&  9 && 14   & \cr
\smalline\hrline\smalline
& 11  && 22 &&  1 && 18 &&  3   & \cr
\smalline\hrline\smalline
& 6   && 17 && 20 && 13 &&  8 & \cr
\smalline\hrline\smalline
& 21  && 12 &&  7 &&  2 && 19 & \cr
\smalline\hrline
}}$$

\noindent we decide to run the tour
again to duplicate that result as well.  We decline the third test in
Table 3.1, because it requires a different value of |max|, which we cannot
change at run-time as the program is currently designed.
Before rerunning the test, we must remember to
```

```
reinitialize the board.  This is the reason that the board initialization
code was separated from all of the other initializations; it is executed
more than once.
We use the same name when defining
the code for this section as for the previous section,
which will cause the second test to follow
immediately after the first in the Pascal program.
@^Knight's Tour@>
@<Perform the Knight's Tour and print the results@>=
@<Initialize the Knight's Tour board@>;
do_the_tour_starting_at(3,3) ;
@<Print the results of the Knight's Tour@>;
@ The 1986 edition shows yet another solution, with (2,4) as the
starting move.


$$\vbox{
\offinterlineskip
\halign{ \vrule # & \strut\  # \ & \vrule # & \  # \  &
      \vrule # & \  # & \vrule # & \  # & \vrule # & \  # & \vrule # \cr
\hrline\smalline
& 23  && 4  && 9  && 14 && 25 & \cr
\smalline\hrline\smalline
& 10  && 15 && 24 &&  1 && 8    & \cr
\smalline\hrline\smalline
& 5   && 22 &&  3 && 18 && 13   & \cr
\smalline\hrline\smalline
& 16  && 11 && 20 && 7  &&  2 & \cr
\smalline\hrline\smalline
& 21  && 6  && 17 && 12 && 19 & \cr
\smalline\hrline
}}$$

@<Perform the Knight's Tour and print the results@>=
@<Initialize the Knight's Tour board@>;
do_the_tour_starting_at(2,4) ;
@<Print the results of the Knight's Tour@>;
@* The output phase.
The job of printing is not as interesting as the problem itself, but it must be
done sooner or later, so we may as well get it over with.


@ In order to keep this program reasonably free of notations that
are uniquely Pascalesque, a few macro definitions for low-level output
instructions are introduced here.  All of the output-oriented commands
in the remainder of the program will be stated in terms of three
simple primitives called |new_line|, |print_string|, and
|print_integer|.
@^system dependencies@>

@d width=3 {width of an integer field}
@d print_string(#)==write(#); {put a given string into the |output| file}
@d print_integer(#)==write(#:width) {print an integer of |width| character
   positions}
```

```
@d new_line==writeln {advance to a new line in the |output| file}

@ @<Print the results of the Knight's Tour@>=
if successful then
   for i := 1 to max do
       begin @/
         for j := 1 to max do
           print_integer(board[i,j]) ;
         new_line;
       end
else
   print_string('no solution') ;
@.no solution@>
new_line;
new_line;
@ We define a few more odd variables.
@<Variables of the program@>=
@!i,@!j : index ; {temporary index variables}
@!successful : Boolean ; {The Knight's Tour was successful }
@* Index.
```

Sample Execution of *knights.p*

```
 1  6 15 10 21
14  9 20  5 16
19  2  7 22 11
 8 13 24 17  4
25 18  3 12 23


23 10 15  4 25
16  5 24  9 14
11 22  1 18  3
 6 17 20 13  8
21 12  7  2 19


23  4  9 14 25
10 15 24  1  8
 5 22  3 18 13
16 11 20  7  2
21  6 17 12 19
```

WEBmeter Generated Output

```
for INPUT file : knights.web

TIDENT     :  140
TNUM       :   59

OPERATORS

and        :    2
array of   :    2
begin end  :    5
boolean    :    2
if then el :    2
for do     :    4
if then    :    3
integer    :    2
not        :    1
or         :    1
procedure  :    1
program    :    1
repeat unt :    1
to         :    4
var        :    1
write      :    2
writeln    :    1
+          :    4
-          :   10
( )        :   29
=          :    5
,          :   16
.          :    1
<          :    1
;          :   64
[ ]        :   24
:=         :   30
..         :    5
<=         :    2
:          :   10
KNIGHTSTOU :    1
INCR       :    1
DECR       :    0
EMPTY      :    4
VALIDROWOR :    2
TARGETPOSI :    1
TARGETSQUA :    1
```

```
TARGETEMPT :    1
SETSQUARET :    2
NUMBEROFLE :    3
TRYKNIGHTM :    2
NOMORECAND :    1
BOARDNOTFU :    1
RECORDMOVE :    1
NEXTMOVE   :    1
ERASEMOVE  :    1
DOTHETOURS :    3
WIDTH      :    1
PRINTSTRIN :    1
PRINTINTEG :    1
NEWLINE    :    3

eta 1                       :    51
total number of operators :   268


OPERANDS

OUTPUT     :    1
1          :   19
MAX        :    7
5          :    3
NUMBEROFSQ :    3
25         :    1
INDEX      :    5
0          :    3
BOARD      :    5
I          :    5
J          :    5
TARGROW    :    5
TARGCOL    :    5
TARGETSQUA :    1
8          :    3
CANDIDATEM :    6
MOVENUMBER :    4
ROW        :    2
COL        :    2
SUCCESSFUL :    9
FALSE      :    1
TRUE       :    1
ROWDELTAS  :   10
COLDELTAS  :   10
-2         :    5
2          :    9
3          :    5
-1         :    4
4          :    3
```

```
6           :    2
7           :    2
POSITIONS   :    1
'no solut'  :    1

eta 2                         :    33
total number of operands      :   148

 SUMMARY

numbered code sections    28
number of procedures       1
number of functions        0

lines  of limbo   14
lines  of doc    263
  per code sect.  9.39
number of macros  19
lines  of code   110
  per code sect.  6.11

VG        :    13.00

eta1      :    51.00
eta2      :    33.00
n1        :   268.00
n2        :   148.00

length    :   416.00
volume    :  2659.20
effort    : 304116.24
time      : 16895.35 seconds
          :   281.59 minutes
          :     4.69 hours


WEB Command Counts

| |         :   70
'...'       :    1
{ }         :    9
#           :   18
==          :   16
=           :    3
@< @>*;     :   17
@< @>=      :   18
@           :    2
@b          :   24
@*          :    4
@d          :   19
```

```
@p          :   1
@<  @>      :  35
@^  @>      :  10
@.  @>      :   2
@!          :  18
@/          :   3
@|          :  20
@;          :   3
```

APPENDIX D

HAND-CALCULATED METRICS

DESIGN and INTEGRATION COMPLEXITY

The design and integration complexity metrics were hand-calculated from the TANGLEd versions of the WEB programs, using the algorithm mentioned in Section 3.1.1.

sample.p $n = 3$

$iv(maxmin) = 1 \quad S_0(maxmin) = 1 \quad iv(mean) = 1 \quad S_0(mean) = 1$

$iv(main) = 1 \quad S_0(main) = iv(main) + S_0(maxmin) + S_0(mean) = 3$

$S_1 = 3 - 3 + 1$

queens.p $n = 2$

$iv(tryqueenmove) = 1 \quad S_0(tryqueenmove) = 1$

$iv(main) = 1 \quad S_0(main) = iv(main) + S_0(tryqueenmove) = 2$

$S_1 = 2 - 2 + 1$

knights.p $n = 2$

$iv(tryknightmove) = 1 \quad S_0(tryknightmove) = 1$

$iv(main) = 1 \quad S_0(main) = iv(main) + S_0(tryknightmove) = 2$

$S_1 = 2 - 2 + 1$

primes.p $n = 1$

$iv(main) = 1 \quad S_0(main) = iv(main) = 1$

$S_1 = 1 - 1 + 1$

reg.p $n = 9$

$iv(blankcoeff) = 1 \quad S_0(blankcoeff) = 1 \quad iv(length) = 1 \quad S_0(length) = 1$

$iv(needparen) = 1 \quad S_0(needparen) = iv(needparen) + S_0(length) = 2$

$iv(orop) = 2 \quad S_0(orop) = iv(orop) + S_0(blankcoeff) + S_0(length) = 4$

$iv(closure) = 3$

$S_0(closure) = iv(closure) + S_0(blankcoeff) + S_0(length) = 5$

$iv(concat) = 6$

$S_0(concat) = iv(concat) + iv(blankcoeff) + iv(length) + iv(needparen) = 9$

$iv(getmat) = 1 \quad S_0(getmat) = 1$

$iv(writematrix) = 1$

$S_0(writematrix) = iv(writematrix) + S_0(getmat) = 2$

$iv(main) = 9$

$S_0(main) = iv(main) + iv(orop) + iv(closure) + iv(concat) + iv(blankcoeff) + iv(getmat) + iv(writematrix) = 23$

$S_1 = 23 - 9 + 1 = 15$

DESIGN STABILITY

The design stability metrics were hand-calculated from the TANGLEd versions of the WEB programs, using the algorithm mentioned in Section 3.2.1.

sample.p $PDS = \frac{1}{17}$

$DLRE_{main} = 10 \quad DS_{main} = \frac{1}{11} \quad DLRE_{maxmin} = 5 \quad DS_{maxmin} = \frac{1}{6}$

$DLRE_{mean} = 1 \quad DS_{mean} = \frac{1}{2}$

queens.p $PDS = \frac{1}{21}$

$DLRE_{main} = 11 \quad DS_{main} = \frac{1}{12} \quad DLRE_{tryqueenmove} = 9 \quad DS_{tryqueenmove} = \frac{1}{10}$

knights.p $PDS = \frac{1}{16}$

$DLRE_{main} = 11 \quad DS_{main} = \frac{1}{12} \quad DLRE_{tryknightmove} = 4 \quad DS_{tryknightmove} = \frac{1}{5}$

primes.p $PDS = 1$

reg.p $PDS = \frac{1}{96}$

$DLRE_{main} = 30 \quad DS_{main} = \frac{1}{31} \quad DLRE_{blankcoeff} = 12 \quad DS_{blankcoeff} = \frac{1}{13}$

$DLRE_{orop} = 4 \quad DS_{orop} = \frac{1}{5} \quad DLRE_{closure} = 4 \quad DS_{closure} = \frac{1}{5}$

$DLRE_{concat} = 6 \quad DS_{concat} = \frac{1}{7} \quad DLRE_{getmat} = 16 \quad DS_{getmat} = \frac{1}{17}$

$DLRE_{writematrix} = 11 \quad DS_{writematrix} = \frac{1}{12} \quad DLRE_{length} = 8 \quad DS_{length} = \frac{1}{9}$

$DLRE_{needparen} = 4 \quad DS_{needparen} = \frac{1}{5}$

VITA

Lisa Min-yi Chen Smith

Candidate for the Degree of

Master of Science

Thesis: MEASURING COMPLEXITY AND STABILITY OF WEB PROGRAMS

Major Field: Computer Science

Biographical:

Personal Data: Born in Louisville, Kentucky, January 24, 1966, the daughter of Boris Y. and Linda L.H. Chen. Married to Gary Wayne Smith on August 5, 1989.

Education: Graduated from Lafayette Senior High School, Lexington, Kentucky in May, 1982; received Bachelor of Science from the University of Kentucky, Lexington, Kentucky, in May, 1986; completed requirements for the Master of Science degree at Oklahoma State University in December, 1990.

Professional Experience: Research Assistant, Applied Research Laboratory, Pennsylvania State University, September 1986 to September 1987. Engineer, General Dynamics – Fort Worth Division, Fort Worth, Texas, October 1987 to August 1988. Graduate Teaching Assistant, Department of Computer Science, Oklahoma State University, August 1989 to May 1990.