REPRINTED FROM (with entirely new fonts and thus different layout)(Original article occupied pages 307–321):
Methodologies for Computer Systems Design
W.K. Giloi and B.D. Shriver (Editors)
Elsevier Science Publishers B.V. (North-Holland)

On the Application of Ada™ and its Tools to the Information Hiding Decomposition Methodology for the Design of Software Systems

Daniel M. Berry*
Computer Science Department
University of California
Los Angeles, CA 90024
U. S. A.

Abstract: This paper shows how Ada can be used as the language for expressing decomposition and interface decisions for software systems designed by application of modern software design methodologies such as Parnas's information hiding methodology. It shows by use of Parnas's KWIC example how most of the required design documents can be written in Ada, how Ada tools can be used to check these documents, and how implementations can be controlled by use of Ada notation.

1 Introduction

Ada [ADA82] was designed to support the programming of embedded systems [DOD78], that is, software which becomes an integral part of the real-world system which it is to control [Leh83]. During the design of Ada, a serious effort was made to address the programming methodologies by which these embedded systems would be designed and implemented. Indeed one of the prime impetuses for the design of a common language was the recognition that widespread use of one language would create a large market for the variety of language-particular tools needed to carry out the modern programming methodologies. The result would be to make production of these tools economically feasible and to eventually permit larger numbers of programmers to be programming with state-of-the-art methodologies and tools. Consequently, Ada contains a number of features which help to express those aspects of software that are manipulated by the design methodologies, i.e., modules. In addition, implementations of the language are to be distributed accompanied by an Ada Programming Support Environment (APSE) [DOD80] which is to provide many tools directly useable in these methodologies.

There is a wide variety of methodologies that are in use and that are envisioned as being used in developing the embedded systems to be written in Ada. Two reports commissioned by the Department of Defense list and evaluate many of these methodologies [FW82, UK81]. It is not a goal of this paper to chose a ''best'' methodology — there is none. Rather, the purpose of this paper is to demonstrate how Ada and some of its envisioned tools may be put to use in applying these methodologies. It is the final contention of this paper that the use of Ada is not restricted to any one methodology and can be used profitably with any of them. However, due to space limitations, that even a book will not solve, this demonstration is carried out using one particular well-known methodology, David Parnas's information hiding methodology [Par72a, b, c]. This methodology is selected as the discussion vehicle, because it was one of the first methodologies to be described in the literature, many other methodologies have the same goals and similar methods, and the main paper describing the methodology is widely read, reprinted, and cited.

The information hiding methodology gives criteria for decomposing a system into modules. The methodology recognizes, as do many others, that the **heavy** cost of any software development project is in carrying out the inevitable maintenance and modification resulting from the discovery of bugs and the desires to enhance the function of the software. The methodology tries to obtain a decomposition into modules such that for any change, the code that must be modified is limited to at most one module. Thus the criteria for

---

™ Ada is a trademark of the U. S. Dept. of Defense (AJPO).

* This work was supported in parts by the Department of Energy, Contract No. DE-AS03-76F0034 P. A. No. DE-AT0376, ER70214, Mod. A006, The University of California MICRO Program, Hughes Aircraft Corporation, SDC — A Burroughs Company, and IBM Corporation.

measuring the goodness of a proposed decomposition is in how well the decomposition isolates changes. The idea is *information hiding*; put in one module *all and only* the code that deals with *one and only one* abstraction, procedural or data, and then *hide* that code inside a module which exports only the abstraction.

In carrying out the creative part of this and any methodologies, there are a number of specifications to be written to serve as blueprints for the later implementation stages. From these specifications, a number of reports are to be generated. Furthermore, there are a number of checks to be done to these specifications in order to minimize the chances that serious design flaws, e.g., interface errors, survive into the implementation stages. It turns out that many of these specifications can be carried out in Ada and a number of the checks can be carried out using existing, already envisioned, and proposed Ada language tools.

In order to demonstrate the application of Ada and its tools to the information hiding methodology, this paper pretends to go through the design of the KWIC Index System that is described in Parnas's two 1972 papers [Par72a, b] (This design is described also in a tutorial article written by Austin Maher [Mah??].). At various points in the design process, it is useful to express the design decision just made or to produce a required document using Ada. At these points, this paper shows the suggested Ada text and indicates how the Ada tools can be used to do some of the checking that should be done in applying the methodology. Also this paper shows how some of the other suggested documents are related to the Ada documents. In doing these, the paper points out what some requirements that the Ada tools should satisfy in order that they be profitably used in the development of software according to information hiding or any of the modern programming methodologies.

2 The Problem

It is assumed that the reader has read at least the paper ''On the Criteria to be Used in Decomposing Systems into Modules'' and is familiar with the conclusions of the paper.

The problem to solve is given in a very short requirement which is paraphrased here:

> The KWIC index system accepts an ordered list of lines; each line is an ordered list of words; and each word is an ordered list of characters. Any line may be circularly shifted by repeatedly removing the first word and appending at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

Appendix I shows a sample page from the output of such a KWIC system.

Parnas presents two decompositions for this problem and determines that the second does a better job of hiding implementation details. As a number of changes to implementation details are contemplated, the effects of each change is confined to one and only one module in the second decomposition. In the first decomposition the effects are much more wide-spread, in some cases all modules have to be changed.

This paper pretends that the two decompositions represent two consecutive attempts to decompose the system correctly and that the second is arrived at after having laid out the first and having analyzed its weaknesses.

3 First Decomposition

Parnas's description of the first decomposition is found in Appendix II. Observe that the sample output in Appendix I resembles that produced by a so-called ''sophisticated system''. Parnas then says ... ''It should be clear that the above does not constitute a definitive document. Much more information would have to be supplied before work could start. The defining documents would include a number of pictures showing core formats, pointer conventions, calling conventions, etc. All of the interfaces between the four modules must be specified before work could begin.'' Apart from the pictures, the definitive document can be written in Ada so that an Ada compiler can be used to check consistency in the choices in and their expressions. Appendix III shows a core diagram showing the array of words generated by module 1 across the top and the various index table arrays produced by 1, 2, and 3 below that.

Appendix IV shows a collection of Ada module skeletons expressing the decisions made on the data structures (the core formats and the pointer conventions), the calling conventions, and the interfaces between the modules. In this Ada text, the data structures declared in the main procedure are those of the diagram. Integers are used as pointers into the *words* array. The procedures are completely parameterless, and the data structures are made available to the procedures via normal global variable access. Note that the document suggests the possibility of separate compilation. Submitting this Ada skeleton to an Ada compiler (with no code generation required) serves to have it completely type and interface checked. The particular document shown passes all tests with no difficulty using the NYU Ada/Ed compiler [NYU81]. Thus the designer is confident that when the modules are finally written, they will fit together properly.

However, the designer is dissatisfied with the decomposition because all data are global variables. Passing data as parameters is supposed to be better. Appendix V shows another Ada program skeleton expressing a decomposition with the same data structures declared in the main procedure, but with all data passed as parameters to all other procedures. Observe which parameters are **in** and which are **out** in each procedure. A datum created by a procedure is made an **out** parameter, and a datum previously created by another procedure is made an **in** parameter. This Ada text is completely type and interface checked by the Ada compiler. Comparing the two versions shows that the advice about preferability of globals notwithstanding, the program that has all procedures accessing all of its data as globals is cleaner.

Appendix VI gives a Stevens, Myers, and Constantine [SMC74] diagram for the decomposition shown in Appendix IV. This diagram is directly deducible from the partial ordering induced by the separate compilation directives in the program skeleton.

4 Second Decomposition

At this point the designer thinks about the ‘‘number of design decisions which are questionable and likely to change under many circumstances’’, and deduces that in fact both realizations of the first decomposition are bad. For example, a change to the location or format of the words causes all modules to be rewritten. As a consequence the designer comes up with the second decomposition. Parnas’s description is given in Appendix VII. The modules which consist ‘‘of a number of subroutines or functions which provide the means by which the user of the module may call on it’’ are recognized as nothing but packages.

The ‘‘more definitive documents’’ describing this second decomposition are the Ada text in Appendix VIII, the Stevens, Myers, and Constantine diagram in Appendix IX, and the specifications given in [Par 72a]. The Ada text consists of a collection of Ada package specification parts and procedure skeletons expressing this decomposition. This Ada code passes through the compiler with no type and interface errors. Thus the designer is certain that there will not be any future interfacing surprises. The diagram of Appendix IX can be deduced from the separate compilation partial order induced by the **with**s and the **separate**s The notation for packages used here is new. The box for a package contains in the top half one sub box for each visible identifier offered by the package, e.g. type, constant, procedure, and function names, and the bottom half contains the name of the whole package module. A directed arc may be directed at a specific sub box (indicated by an arrow head on the arc) or at the whole box (indicated by an asterisk on the arc) according to the level of the decomposition.

Consider the behavioral specifications of [Par 72a]. Given that these specifications were written long before Ada was even a green idea, it is amazing how well they match the packages. The sections of the behavioral specifications correspond one-for-one with the packages, The (non-mapping and thus externally visible) functions of the behavioral specification correspond one-for-one with the exported subprograms of the packages. Of course, the packages were designed with this correspondence in mind; the point is that it is so easy to preserve this correspondence. In fact, one could attach each piece of the behavioral specification as a comment defining the semantics of the corresponding Ada text. This commentary thus provides semantic specification of the packages and serves as a contract for the implementors and users of the packages.

Parnas compares the two decompositions by saying that ... ‘‘Both schemes will work. The first is quite conventional [at least it was in 1972], the second has been used successfully in a class project ... [That is, each module was assigned to different people to be programmed and tested separately. Then complete programs were formed out of different combinations of the separate modules. About 20% of the complete programs worked reflecting that only about 20% of the individual modules failed to work properly [Par72c].] Both will reduce the programming to the relatively independent programming of a number of small, manageable, programs.’’

What Parnas is claiming is that the specification given so far for the second decomposition is sufficient for separate people to go off and separately program each piece and expect that the pieces will fit together. Each programmer can chose his or her own implementation for the data structures of his or her package. This property does not hold for the first decomposition. That decomposition requires explicit agreement as to the implementing data structures before each piece can be given to separate programmers. Imagine the chaos if each programmer decided on the data structures for his or her own module. Thus in the first case, the presence of data structure declarations in the ''more definitive documents'' is critical. In second case, the document is sufficiently definitive without any data structure declarations. This difference is what is meant by the claim that the second decomposition is doing a better job of hiding implementation details. Another way to see this is that in the first decomposition, the data structures are part of the interface, but in the second they are not even visible in the parts defining the interface.

4 Improvements

Now closer examination of the skeletons show that the operations offered by the *CIRCULAR_SHIFTS* package are not sufficient to produce the sophisticated output such as shown in Appendix I. The call to *ith* yields a circular shift line index. However, that line is to be printed in its original order with the first word of the circular shift in the center of the output page. The first word of the circular shift is easy to identify with the given operations, but there is no way to identify the original line or its first word (either suffices). Thus at least one operation to do either of the above is necessary. In order to allow either, the designer adds both. The first is an function *line_index* such that *line_index(i)* is the index of the line whose circular shift is the *i*th. The second is *shift* such that *shift(i)* is the number of words that were circularly shifted to arrive at the *i*th circular shift line. Interestingly, these were already internal (mapping) functions of Parnas's specification, but with the names HIP and SHI respectively. These improvements are included in the Ada code of Appendix XI. The improvements are in the lines marked by a vertical bar on the left margin. This code also went through the Ada compiler with no type or interface errors.

5 Optimization of Implementations

Parnas continues his analysis. ''Note first that the two decompositions may share all data representations and access methods. Our discussion is about two different ways of cutting up what *may* be the same object. A system built according to decomposition 1 could conceivably be identical *after assembly* to one built according to decomposition 2. The differences between the two alternatives are in the way that they are divided into work assignments, and the interfaces between modules. The algorithms used in both cases *might* be identical.''

One of the problems with the second decomposition is that it could have a very poor performance. Practically everything is done with a procedure call. In fact what the first decomposition does by a direct access to a particular element of one of the memory arrays the second does by a call to a function whose only effect is to return the value of the particular array element. This call overhead can be eliminated if procedure calls are expanded in-line. This in-line expansion can be done by the use of an editor working on the program text prior to a compilation or by use of a language feature that specifies in-line compilation. The latter is to be preferred because then the program text never changes from the fully modular form. That is the modules can be left in the best form for dealing with the later modifications and enhancements gracefully, while at any time code can be generated which is in the best form for performance.

Thus, what Parnas is referring to is that by judicious choice of data structures and in-line expansion of procedure calls, the code for the two decomposition can be made identical. So the designer takes the decomposition of Appendix VIII together with the improvements of section 4 and adds bodies to the packages giving data structures to implement the abstractions. The data structures chosen are precisely those of the first decomposition. (The variable *words* has to be changed so that it does not clash with the function *words* returning the number of words in a given line. The variable is now called *the_words*.) Note that *the_words* array and the *line_index* table are in the body of the *LINE_STORAGE* package, the *cs_line_index* table is in the body of the *CIRCULAR_SHIFTS* package, and the *alphed_cs_line_index* table is in the body of the *ALPHABETIZER* package. In addition, a number of the procedures are given the **pragma** *IN_LINE* in order to indicate to the compiler that calls to them are to be expanded in line. If the compiler obeys this **pragma**, there is no calling overhead in each case. The procedures selected for this treatment are those whose bodies are likely to be small and which are doing the direct accesses to the arrays that occur in the first decomposition. The resulting Ada code is in Appendix XI. Observe that the package bodies contain dummy bodies for each of the procedures; without this the compiler refuses to accept the code and does none of the

desired type and interface checking.

The semantics of Ada say that the *IN_LINE* **pragma**, as any other **pragma**, does not have to be obeyed by the compiler, and in fact the compiler used to check the programs in this paper does not. When the latest Ada code is put through the compiler, there are six semantic errors, all of them complaining that the compiler does not accept the *IN_LINE* **pragma**.*

In any case, it is clear that by playing with the bodies of the procedures that are to be compiled in-line and with which procedures are to be compiled in-line, one can get the code generated by the second decomposition to be as close to that of the first as desired. It is interesting to see that in-line expansion has the effect of bringing a data structure declared inside one module into another module to be accessed as if it were declared inside the other module. For the programmer to do so would be a serious breach of modularity and a scope error as well. However, once the compiler has verified that the programmer has done nothing to violate scope and thus nothing to violate modularity, it can go ahead and violate both left and right to achieve efficiency!

6 Exceptions

Parnas gives for each module a list of exceptions that may be raised if certain conditions are not met on the call of the various functions of the modules. These exceptions amount to constraint errors on the abstraction. These exceptions can be expressed as Ada exceptions. Appendix XII shows the exception declarations to be added to each package. Then the callers of the various routines may provide handlers to respond to the conditions as Parnas suggests should be done.

7 Makefile

From the separate compilation partial order given in the Ada code, it is possible to derive a *makefile* [Fel78] which can be used to cause recompilation of only the parts affected by a given change. The makefile resulting from the code of Appendix XI is given as Appendix XIII. The makefile assumes that the specification part and the body of a package occupy the same file.

8 Conclusions

This paper has gone through a simulated top-level design using Parnas's information hiding methodology. Ada was used as the means to express design decisions and as the medium for giving much of the definitive documentation needed to be able to carry out the work. The human programmers are still required to think and the use of Ada does not reduce that need. What the use of Ada and its tools does is to provide a means for checking the consistency of the design decisions and their expression to the extent possible without going so far as formal verification. With the type and interface checking that the Ada compiler can do, the designer is certain the the modules fit together and that major interface errors will not show up later. It is unfortunate and stupid if easily detected error remain in a design to later completely invalidate what would otherwise be a good design.

Ada has been used more as a *module interconnection language (MIL)* [DK76, Tho76, Pen81, PB79, PBE81] rather than as a *program design language* [CFG75, CG75]. The modular decomposition and interface have been precisely specified while the contents of the modules have been left mostly unspecified. In the case of a MIL, the precision of full Ada is appreciated and is in fact needed in order that an Ada compiler do the proper checking. It is this author's belief, the existing examples to the contrary [Wau80], that full Ada does not make a good program design language. A program design language should permit the use of parameterized natural language sentences with recognition and type checking of declared words that happen to appear as parameters in the midst of these sentences [BYY83]. Neither Ada nor its compilers can support this feature. A properly designed program design language such as SDP [Yav80] is better. SDP has parameterized natural language sentences, optional type checking, and the ability to define abstract data types [LZ74] so that the data type definitions do not end up being too detailed for a design document.

---

* Clearly, the ability to expand procedures in-line is critical to the success of the methodology in producing efficient, useable code. Therefore, this author recommends not buying any compiler that does not accept it. Let the free market forces dictate what **pragma**s are obeyed!

This exercise has pointed out a number of requirements for Ada tools.

1. An option should exist in Ada compilers to allow it to do full type checking when it is presented with packages that have private types but no private parts describing their implementation. The private part is not needed for the type and interface checking and it is nice to be able to check types and interfaces out completely *before* implementations are decided on. Apparently the NYU Ada/Ed compiler incorrectly behaves this way. This is good, but is non-standard and thus can be available only as an option.

2. It seems that the in-line expansion capability is absolutely essential for use of modern design methodologies to produce efficient programs and should not be optional.

3. There should exist tools to check at least the interface consistency of an Ada module and its formal specification. It is stupid not to have checked this and to discover later that the specification is of a close but nevertheless different module. Such a simple checker would help prevent this occurrence. Another idea is to embed the specification into Ada so that the Ada syntax for procedure and package interfaces is accepted as giving the interface of specification procedures and modules. This is the approach of Anna [KL80].

Finally, note that while this paper has demonstrated the use of Ada and its tools with one particular methodology of software system design, it is the author's firm belief that Ada and its tools are also applicable to other methodologies as well. For example, the author has used Ada and its tools with Composite Design and Analysis [SCM74] as well and is incorporating Ada and its tools into the SARA System Design Methodology [Pen79, Raz77, RVB80, RVE79, REFRSV79, Sil81, VOR80, VE84]. The point is that almost all of the methodologies are doing programming-in-the-large [DK76] and as such are dealing with the module as the unit of discourse. Ada has nice linguistic features for dealing with modules and its tools are likewise conversant with modules [Bry84]. This author urges the reader to try using Ada and its tools with his or her favorite methodology.

Bibliography

[ADA82] ''Reference Manual for the Ada Programming Language'', U. S. Department of Defense, MIL-STD-1815, 1982.

[Bry83] ''On the Application of Ada and its Tools to the Information Hiding Decomposition Methodology for the Design of Software Systems'', Computer Science Department, UCLA, August 1983.

[Bry84] Berry, D.M. ''On the Use of Ada as a Module Interconnection Language'', *Proceedings of the Seventeenth Annual Hawaii International Conference on Systems Science*, Honululu, Hawaii, January 1984.

[BYY83] Berry, D.M., Yavne, N., and Yavne, M. ''On the Requirements for a Program Design Language: Parameterization, Abstract Data Typing, and Strong Typing in Software Design Processor (SDP)'', Computer Science Dept., UCLA, 1983.

[CFG75] Caine, S.H., Farber, D.J., Gordon, K.E., ''Program Design Language'', Caine, Farber and Gordon, Inc., March 1975.

[CG75] Caine, S.H., Gordon, K.E., ''PDL - A Tool for Software Design'', *AFIPS Proceedings of the 1975 National Computer Conference*, Vol. 44, May, 1975, pp. 271-276.

[DOD78] ''Department of Defense Requirements for High Order Computer Programming Languages, STEELMAN'', U. S. Department of Defense, 1978.

[DOD80] ''Department of Defense Requirements for Ada Programming Support Environments, STONEMAN'', U. S. Department of Defense, 1980.

[DK76] De Remer, F. and Kron, H. H. ''Programming-in-the-Large Versus Programming-in-the-Small'', *IEEE Trans. on SE*, SE-2:2, June, 1976.

[Fel78] Feldman, S. I., ‘‘Make — A program for Maintaining Computer Programs’’, Bell Laboratories, Murray Hill, NJ, 1978.

[FW82] Freeman, P., Wasserman, A. I., ‘‘Software Development Methodologies and Ada, METHODMAN’’, U. S. Department of Defense, AJPO, 1982.

[Leh83] Lehman, M. M., Private Communication, 1983.

[LZ74] Liskov, B.H. and Zilles, S. N., ‘‘Programming with Abstract Data Types’’, *ACM SIGPLAN NOTICES*, Vol. 9, No. 4, April 1974, pp. 50-60.

[KL80] Krieg-Brückner, B. and Luckham, D. C., ‘‘ANNA: Towards a Language for Annotating Ada Programs’’, *Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language, SIGPLAN Notices*, Vol. 15, No. 11, November, 1980, pp. 128-138.

[Mah??] Maher, A. ‘‘Parnas Decomposition’’, Kearfott Division, Singer, Wayne, NJ, 19??.

[NYU81] ‘‘The NYU Ada/Ed System, An Overview’’, Courant Institute, New York University, 1981.

[Par72a] Parnas, D. L., ‘‘A Technique for Software Module Specifications with Examples’’, *Comm. ACM*, Vol. 15, No. 5, May, 1972, pp. 330-336.

[Par72b] Parnas, D. L., ‘‘On the Criteria to be Used in Decomposing Systems into Modules’’, *Comm. ACM*, Vol. 15, No. 12, Dec., 1972, pp. 1053-1058.

[Par72c] Parnas, D. L., ‘‘Some Conclusions from an Experiment in Software Engineering Techniques’’, *Proceedings of the FJCC*, Vol. 41, 1972, pp. 325-329.

[Pen79] Penedo, M. H., ‘‘SL1 System Reference Manual’’, Internal Memorandum #190a, Revised UCLA, August 1979.

[Pen81] Penedo, M. H., ‘‘The Use of a Module Interconnection Description in the Synthesis of Reliable Software Systems’’, Ph. D. Dissertation, UCLA, Report No. CSD-810115, January, 1981.

[PB79] Penedo, M.H. and D. Berry, ‘‘The Use of a Module Interconnection Language in the SARA System Design Methodology’’, *Proceedings of the 4th International Conference on Software Engineering*, Munich, Germany, September 1979.

[PBE81] Penedo, M., D. Berry and G. Estrin, ‘‘An Algorithm to Support Code-Skeleton Generation for Concurrent Systems’’, *IEEE 5th International Conference on Software Engineering*, March 1981.

[Raz77] Razouk, R., ‘‘The GMB Simulator System Reference Manual’’, UCLA, July 27, 1977.

[RVB80] Razouk, R., M. Vernon and M. Brewer, ‘‘Control-Flow Analyzer Reference Manual’’, UCLA, February 8, 1980.

[RVE79] Razouk, R., M. Vernon and G. Estrin, ‘‘Evaluation Methods in SARA -- The Graph Model Simulator’’, *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems*, August 1979, pp. 189-206.

[REFRSV79] Ruggiero, W., G. Estrin, R. Fenchel, R. Razouk, S. Schwabe and M. Vernon, ‘‘Analysis of Data Flow Models Using the SARA Graph Model of Behavior’’, *AFIPS Conference Proceedings, National Computer Conference*, Anaheim, CA, June 1979.

[Sil81] Silva, E. ‘‘New User’s Guide’’, Internal Memorandum #205, UCLA, January 1981.

[SMC74] Stevens, W. P., Myers, G. F., and Constantine, L. L., ‘‘Structured Design’’, *IBM Systems Journal*, Vol. 13, No. 2, 1974, pp. 115-139.

[Tho76] Thomas, J. W., ‘‘Module Interconnection in Programming Systems Supporting Abstraction’’, Ph. D. Dissertation, Brown University, Providence, RI, April, 1976.

[UK81] Easteal, B. R., Pickett, M. J., Denvir, B. T., Jackson, M. I., Dignan, A. J., Taylor, W. J., Davis, N. W., Tate, A. R., and Harwood, W., ‘‘Report of the Study of an Ada Based System Development Methodology’’, U.K. Department of Industry, 1981.

[VOR80] M. Vernon, W. Overman and R. Razouk, ‘‘GMB PL1 Preprocessor Reference Manual’’, UCLA, January 18, 1980.

[VE83] Vernon, M. and Estrin, G. ‘‘The UCLA Graph Model of Behavior: Support for Performance-Oriented Design’’, These proceedings, 1984.

[Wau80] Waugh, D. W. ‘‘Ada as a Design Language’’, *IBM Software Engineering Exchange* Vol. 3, No. 1, October, 1980.

[Yav80] Yavne, N. ‘‘Software Development Processor User Reference Manual’’, Mayda Software Engineering, P. O. B. 1389, Rehovot, Israel, 1980.

Appendix I

Make	- A Program for Maintaining Computer Programs	[Fel78]
Make -	A Program for Maintaining Computer Programs	[Fel78]
Reference Manual for the	Ada Programming Language	[ADA81]
The NYU	Ada/Ed System, An Overview	[NYU81]
The NYU Ada/Ed System,	An Overview	[NYU81]
Make - A Program for Maintaining	Computer Programs	[Fel78]
Program	Design Language	[CFG75]
Software	Development Processor User Reference Manual	[Yav80]
Make - A Program	for Maintaining Computer Programs	[Fel78]
Reference Manual	for the Ada Programming Language	[ADA81]
The	Ina Jo Reference Manual	[LSSE80]
The Ina	Jo Reference Manual	[LSSE80]
Program Design	Language	[CFG75]
Reference Manual for the Ada Programming	Language	[ADA81]
Make - A Program for	Maintaining Computer Programs	[Fel78]
	Make - A Program for Maintaining Computer Programs	[Fel78]
Software Development Processor User Reference	Manual	[Yav80]
The Ina Jo Reference	Manual	[LSSE80]
Reference	Manual for the Ada Programming Language	[ADA81]
The	NYU Ada/Ed System, An Overview	[NYU81]
The NYU Ada/Ed System, An	Overview	[NYU81]
Software Development	Processor User Reference Manual	[Yav80]
	Program Design Language	[CFG75]
Make - A	Program for Maintaining Computer Programs	[Fel78]
Reference Manual for the Ada	Programming Language	[ADA81]
Make - A Program for Maintaining Computer	Programs	[Fel78]
Software Development Processor User	Reference Manual	[Yav80]
The Ina Jo	Reference Manual	[LSSE80]
	Reference Manual for the Ada Programming Language	[ADA81]
	Software Development Processor User Reference Manual	[Yav80]
The NYU Ada/Ed	System, An Overview	[NYU81]
Reference Manual for	the Ada Programming Language	[ADA81]
	The Ina Jo Reference Manual	[LSSE80]
	The NYU Ada/Ed System, An Overview	[NYU81]
Software Development Processor	User Reference Manual	[Yav80]

Appendix II

Modularization 1:

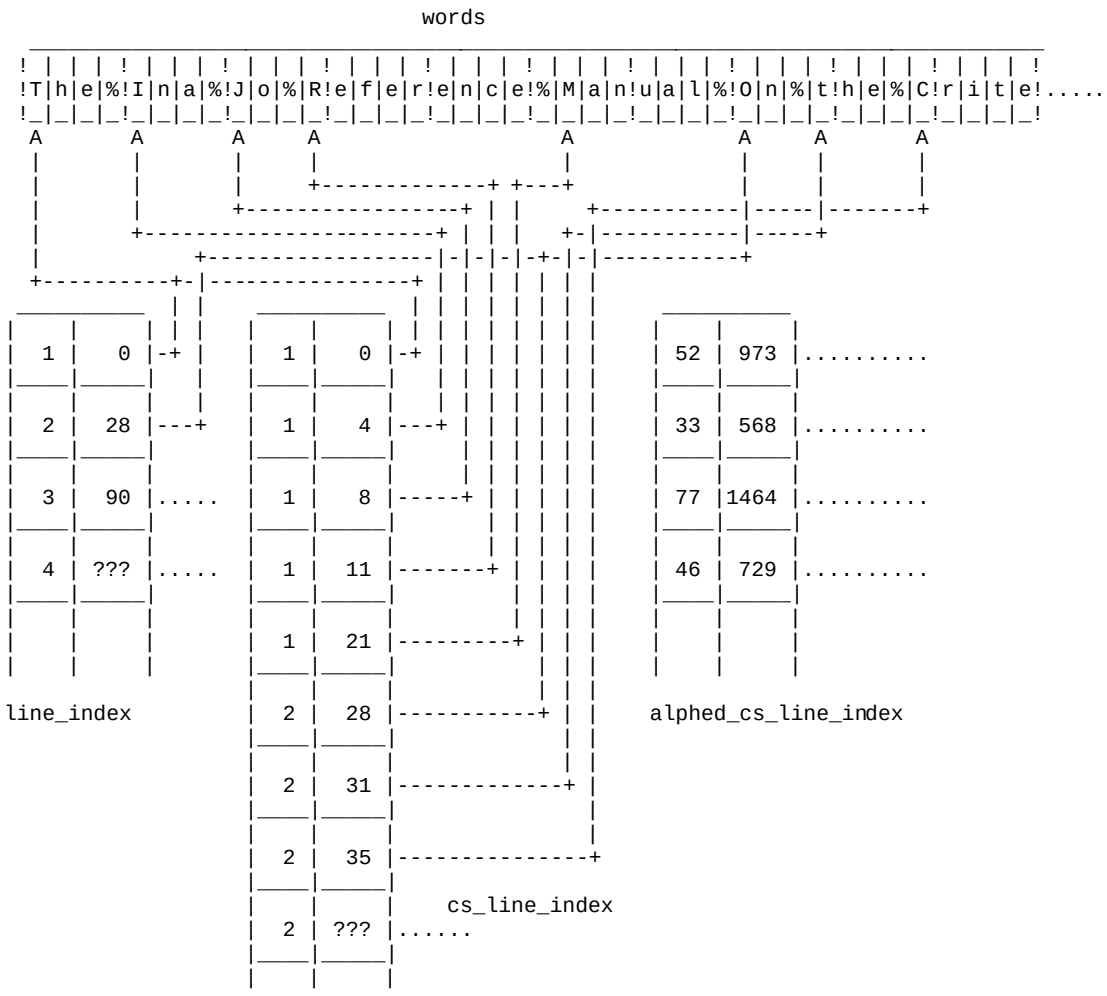
Module 1: Input. This module reads the data lines from the input medium and stores them in core for processing by the remaining modules. The characters are packed four to a word, and an otherwise unused character is used to indicate the end of a word. An index is kept to show the starting address of each line.

Module 2: Circular Shift. This module is called after the input module has completed its work. It prepares an index which gives the address of the first character of each circular shift, and the original index of the line in the array made up by module 1. It leaves its output in core with words in pairs (original line number, starting address).

Module 3: Alphabetizing. This module takes as input the arrays produced by module 1 and 2. It produces an array in the same format as that produced by module 2. In this case, however, the circular shifts are listed in another order (alphabetically).

Module 4: Master Control. This module does little more than control the sequencing among the other four modules. It may also handle error messages, space allocation, etc.

Appendix III



Appendix IV

```
procedure kwic is
    use SYSTEM;
    type PAIR is record
        line_no:INTEGER;
        character_no:INTEGER;
    end record;
    type INDEX_TABLE is array
        (INTEGER range <>) of PAIR;
    no_of_characters: constant INTEGER:=
        MAX_INT;

    words:STRING(1..no_of_characters);
    line_index:INDEX_TABLE(1..no_of_characters);
    cs_line_index:INDEX_TABLE
        (1..no_of_characters);
    alphed_cs_line_index:INDEX_TABLE
        (1..no_of_characters);

    procedure input is separate;
    procedure make_circular_shifts is separate;
    procedure alphabetize is separate;
    procedure output is separate;

begin
```

```
        input;
        make_circular_shifts;
        alphabetize;
        output;

end kwic;

separate(kwic)
procedure input is
begin
        null;
end;

separate(kwic)
procedure make_circular_shifts is
begin
        null;
end;

separate(kwic)
procedure alphabetize is
begin
        null;
end;

separate(kwic)
procedure output is
begin
        null;
end;
```

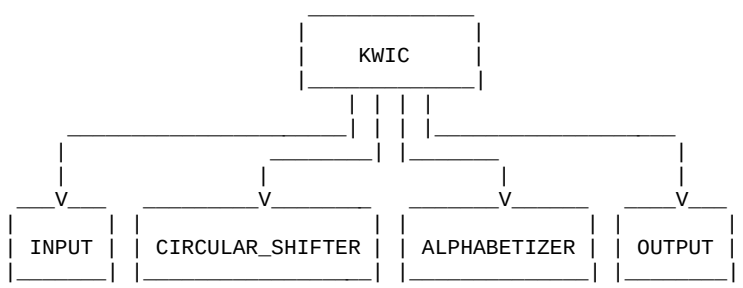
Appendix V

```
procedure kwic is
        use SYSTEM;
        type PAIR is record
                line_no:INTEGER;
                character_no:INTEGER;
        end record;
        type INDEX_TABLE is array
                (INTEGER range <>) of PAIR;
        no_of_characters: constant INTEGER:=
                MAX_INT;

        words:STRING(1..no_of_characters);
        line_index:INDEX_TABLE
                (1..no_of_characters);
        cs_line_index:INDEX_TABLE
                (1..no_of_characters);
        alphed_cs_line_index:INDEX_TABLE
                (1..no_of_characters);

        procedure input(words:out STRING
                        (1..no_of_characters);
                line_index:out INDEX_TABLE
                        (1..no_of_characters))
                is separate;

        procedure make_circular_shifts(words:in
                         STRING(1..no_of_characters);
                line_index:in INDEX_TABLE
                        (1..no_of_characters);
                cs_line_index:out INDEX_TABLE
                        (1..no_of_characters))
                is separate;
```

```
        procedure alphabetize(words:in STRING
                (1..no_of_characters);
            line_index:in INDEX_TABLE
                (1..no_of_characters);
            cs_line_index:in INDEX_TABLE
                (1..no_of_characters);
            alphed_cs_line_index:out INDEX_TABLE
                (1..no_of_characters))
            is separate;

        procedure output(words:in STRING
                (1..no_of_characters);
            line_index:in INDEX_TABLE
                (1..no_of_characters);
            cs_line_index:in INDEX_TABLE
                (1..no_of_characters);
            alphed_cs_line_index:in INDEX_TABLE
                (1..no_of_characters))
            is separate;

begin
    input(words,line_index);
    make_circular_shifts(words,line_index,
        cs_line_index);
    alphabetize(words,line_index,cs_line_index,
        alphed_cs_line_index);
    output(words,line_index,cs_line_index,
        alphed_cs_line_index);

end kwic;

separate(kwic)
procedure input(words:out STRING
                (1..no_of_characters);
    line_index:out INDEX_TABLE
        (1..no_of_characters))
    is
begin
    null;
end;

separate(kwic)
procedure make_circular_shifts(words:in STRING
        (1..no_of_characters);
    line_index:in INDEX_TABLE
        (1..no_of_characters);
    cs_line_index:out INDEX_TABLE
        (1..no_of_characters))
    is
begin
    null;
end;

separate(kwic)
procedure alphabetize(words:in STRING
        (1..no_of_characters);
    line_index:in INDEX_TABLE
        (1..no_of_characters);
    cs_line_index:in INDEX_TABLE
        (1..no_of_characters);
    alphed_cs_line_index:out INDEX_TABLE
        (1..no_of_characters))
    is
begin
    null;
end;
```

```
separate(kwic)
procedure output(words:in STRING
        (1..no_of_characters);
    line_index:in INDEX_TABLE
        (1..no_of_characters);
    cs_line_index:in INDEX_TABLE
        (1..no_of_characters);
    alphed_cs_line_index:in INDEX_TABLE
        (1..no_of_characters))
    is
begin
    null;
end;
```

Appendix VI



Appendix VII

Modularization 2:

Module 1: Line Storage. This module consists of a number of functions or subroutine which provide the means by which the user of the module may call on it. The function call *CHAR(r,w,c)* will have as value an integer representing the *c*th character in the *r*th line, *w*th word. A call such as *SETCHAR(r,w,c,d)* will cause the *c*th character in the *w*th word of the *r*th line to be the character represented by *d* (i.e. *CHAR(r,w,c)=d*). *WORDS(r)* returns as value the number of words in line *r*. There are certain restrictions in the way that these routines may be called; if these restrictions are violated the routines "trap" to an error-handling subroutine which is to be provided by the users of the routine. Additional routines are available which reveal to the caller the number of words in any line, the number of lines currently stored, and the number of characters in any word. Functions *DELINE* and *DELWRD* are provided to delete portions of lines which have already been stored. A precise specification ... has been given in [Par 72a] and we will not repeat it here.

Module 2: INPUT. This module reads the original lines from the input media and calls the line storage module to have them stored internally.

Module 3: Circular Shifter. The principal functions provided by this module are analogs of functions provided in module 1. The module creates the impression that we have created a line holder containing not all of the lines but all of the circular shifts of the lines. Thus the function call *CSCHAR(l,w,r)* provides the value representing the *c*th character in the wth word of the *l*th circular shift. It is specified that (1) if *i<j* then the shifts of line *i* precede the shifts of line *j*, and (2) for each line the first shift is the original line, the second shift is obtained by making a one-word rotation to the first shift, etc. A function *CSSETUP* is provided which must be called before the other functions have their specified values. For a more precise specification of such a module see [Par 72a].

Module 4: Alphabetizer. This module consists principally of two functions. One, *ALPH*, must be called before the other will have a defined value. The second, *ITH*, will serve as an index. *ITH(i)* will give the index of the circular shift which comes *i*th in the alphabetical ordering. Formal definitions of these functions are given [in Par 72a].

Module 5: Output. This module will give the desired printing of set of lines or circular shifts.

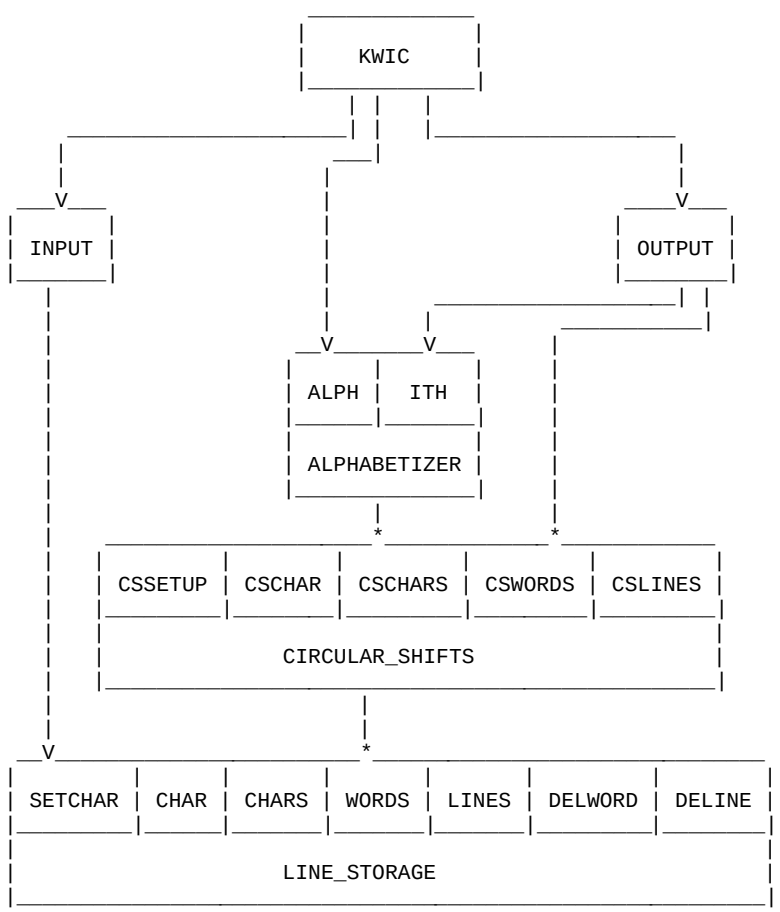
Module 6: Master Control. Similar in function to the modularization above.

Appendix VIII

```
package LINE_STORAGE is
      function char(l,w,c:INTEGER)return
            CHARACTER;
      procedure setchar(l,w,c:INTEGER;
            d:CHARACTER);
      function chars(l,w:INTEGER)return INTEGER;
      function words(l:INTEGER)return INTEGER;
      function lines return INTEGER;
      procedure delword(l,w:INTEGER);
      procedure deline(l:INTEGER);
end LINE_STORAGE;

with LINE_STORAGE;
use LINE_STORAGE;
package CIRCULAR_SHIFTS is
      procedure cssetup;
      function cschar(l,w,c:INTEGER)return
            CHARACTER;
      function cschars(l,w:INTEGER)return INTEGER;
      function cswords(l:INTEGER)return INTEGER;
      function cslines return INTEGER;
end CIRCULAR_SHIFTS;

with CIRCULAR_SHIFTS;
use CIRCULAR_SHIFTS;
package ALPHABETIZER is
      procedure alph;
      function ith(i:INTEGER)return INTEGER;
end ALPHABETIZER;

with ALPHABETIZER;
use ALPHABETIZER;
procedure kwic is
      procedure input is separate;
      procedure output is separate;
begin
      input;
      alph;
      output;
end;

with LINE_STORAGE;
use LINE_STORAGE;
separate(kwic)
procedure input is
      l,w,c:INTEGER;
      d:CHARACTER;
begin
      -- read in characters one by one into d,
      -- breaking into words and lines and setting
      -- the line, word, and character indices
      -- l,w, and c appropriately and do
      setchar(l,w,c,d);
      -- for each
end INPUT;


with ALPHABETIZER,CIRCULAR_SHIFTS,TEXT_IO;
use ALPHABETIZER,CIRCULAR_SHIFTS,TEXT_IO;
separate(kwic)
procedure output is
      l:INTEGER;
begin
```

```
        for i in 1..cslines( ) loop
            l:=ith(i);
            for w in 1..cswords(l) loop
                for c in 1..cschars(l,w) loop
                    -- in the proper place for the fancy
                    -- output do
                    put(cschar(l,w,c));
                end loop;
            end loop;
        end loop;
end OUTPUT;
```

Appendix IX



Appendix XI

```
package LINE_STORAGE is
      p1:constant INTEGER;
      p2:constant INTEGER;
      p3:constant INTEGER;

      function char(l,w,c:INTEGER)return
            CHARACTER;
      procedure setchar(l,w,c:INTEGER;
            d:CHARACTER);
      function chars(l,w:INTEGER)return INTEGER;
      function words(l:INTEGER)return INTEGER;
      function lines return INTEGER;
      procedure delword(l,w:INTEGER);
      procedure deline(l:INTEGER);
|     pragma IN_LINE(char,setchar);
```

```
end LINE_STORAGE;

package body LINE_STORAGE is
      use SYSTEM;
      type PAIR is record
            line_no:INTEGER;
            character_no:INTEGER;
      end record;
      type INDEX_TABLE is array
            INTEGER range <>) of PAIR;
      no_of_characters: constant INTEGER:=
            MAX_INT;

      the_words:STRING(1..no_of_characters);
      line_index:INDEX_TABLE(1..no_of_characters);

      function char(l,w,c:INTEGER)return
            CHARACTER is begin null; end;
      procedure setchar(l,w,c:INTEGER;
            d:CHARACTER) is begin null; end;
      function chars(l,w:INTEGER)return INTEGER
            is begin null; end;
      function words(l:INTEGER)return INTEGER
            is begin null; end;
      function lines return INTEGER
            is begin null; end;
      procedure delword(l,w:INTEGER)
            is begin null; end;
      procedure deline(l:INTEGER) is begin null; end;
      pragma IN_LINE(char,setchar);
end LINE_STORAGE;

with LINE_STORAGE;
use LINE_STORAGE;
package CIRCULAR_SHIFTS is

      p4:constant INTEGER;

      procedure cssetup;
      function line_index(l:INTEGER)return INTEGER;
      function shift(l:INTEGER)return INTEGER;
      function cschar(l,w,c:INTEGER)return
            CHARACTER;
      function cschars(l,w:INTEGER)return INTEGER;
      function cswords(l:INTEGER)return INTEGER;
      function cslines return INTEGER;
|     pragma IN_LINE(cssetup,cschars);
end CIRCULAR_SHIFTS;

package body CIRCULAR_SHIFTS is
      use SYSTEM;
      type PAIR is record
            line_no:INTEGER;
            character_no:INTEGER;
      end record;
      type INDEX_TABLE is array
            (INTEGER range <>) of PAIR;
      no_of_characters: constant INTEGER:=
            MAX_INT;

      cs_line_index:INDEX_TABLE
            (1..no_of_characters);

      procedure cssetup is begin null; end;
      function line_index(l:INTEGER)return INTEGER
            is begin null; end;
```

```
        function shift(l:INTEGER)return INTEGER
            is begin null; end;
        function cschar(l,w,c:INTEGER)return
            CHARACTER is begin null; end;
        function cschars(l,w:INTEGER)return INTEGER
            is begin null; end;
        function cswords(l:INTEGER)return INTEGER
            is begin null; end;
        function cslines return INTEGER
            is begin null; end;
        pragma IN_LINE(cssetup,cschars);

end CIRCULAR_SHIFTS;

with CIRCULAR_SHIFTS;
use CIRCULAR_SHIFTS;
package ALPHABETIZER is
        procedure alph;
        function ith(i:INTEGER)return INTEGER;
|       pragma IN_LINE(ith);
end ALPHABETIZER;

package body ALPHABETIZER is
        use SYSTEM;
        type PAIR is record
            line_no:INTEGER;
            character_no:INTEGER;
        end record;
        type INDEX_TABLE is array
            (INTEGER range <>) of PAIR;
        no_of_characters: constant INTEGER:=
            MAX_INT;

        alphed_cs_line_index:INDEX_TABLE
            (1..no_of_characters);

        procedure alph is begin null; end;
        function ith(i:INTEGER)return INTEGER
            is begin null; end;
        pragma IN_LINE(ith);

end ALPHABETIZER;

with ALPHABETIZER;
use ALPHABETIZER;
procedure kwic is
        procedure input is separate;
        procedure output is separate;
begin
        input;
        alph;
        output;
end;

with LINE_STORAGE;
use LINE_STORAGE;
separate(kwic)
procedure input is
        l,w,c:INTEGER;
        d:CHARACTER;
begin
        -- read in characters one by one into d,
        -- breaking into words and lines and setting
        -- the line, word, and character indices
        -- l,w, and c appropriately and do
        setchar(l,w,c,d);
```

```
      -- for each
end INPUT;


with ALPHABETIZER,CIRCULAR_SHIFTS,
      LINE_STORAGE,TEXT_IO;
use ALPHABETIZER,CIRCULAR_SHIFTS,
      LINE_STORAGE,TEXT_IO;
separate(kwic)
procedure output is
      l,k,v:INTEGER;
begin

      for i in 1..cslines( ) loop
            l:=ith(i);
|           k:=line_index(l);
|           for w in 1..words(l) loop
|                 for c in 1..chars(l,w) loop
|                       v:=shift(l);
|                       -- observing that the v th word is the
|                       -- first word of the kth circular shift
|                       -- which is ith in the alphabetical ordering
|                       -- and is a shift of the l th line,
|                       -- in the proper place for the fancy
|                       -- output do
|                       put(char(l,w,c));
|                 end loop;
|           end loop;
      end loop;

      for i in 1..lines( ) loop
            for w in 1..words(l) loop
                  delword(l,w);
            end loop;
            deline(l);
      end loop;

end OUTPUT;
```

Appendix XII

```
package LINE_STORAGE is
            ...
            ERLGEL:exception;
            ERLGNL:exception;
            ERLGEW:exception;
            ERLGNW:exception;
            ERLGEC:exception;
            ERLGNC:exception;
            ERLSEL:exception;
            ERLSBL:exception;
            ERLSEW:exception;
            ERLSBW:exception;
            ERLSEC:exception;
            ERLSBC:exception;
            ERLCNL:exception;
            ERLCNW:exception;
            ERLWNL:exception;
            ERLDLE:exception;
            ERLDWE:exception;
            ERLDLL:exception;

end LINE_STORAGE;

with LINE_STORAGE;
use LINE_STORAGE;
```

```
package CIRCULAR_SHIFTS is
        ...
        ERCNES:exception;
        ERCIND:exception;
        ERCINL:exception;
        ERCSND:exception;
        ERCSNL:exception;
        ERCGND:exception;
        ERCGNL:exception;
        ERCGNW:exception;
        ERCGNC:exception;
        ERCCND:exception;
        ERCCNL:exception;
        ERCCNW:exception;
        ERCWND:exception;
        ERCWNL:exception;
        ERCLND:exception;

end CIRCULAR_SHIFTS;

with CIRCULAR_SHIFTS;
use CIRCULAR_SHIFTS;
package ALPHABETIZER is
        ...
        ERAEBL:exception;
        ERAEBW:exception;
        ERAWBL:exception;
        ERAWBW:exception;
        ERALEL:exception;
        ERAALB:exception;
        ERAINL:exception;
        ERAIND:exception;

end ALPHABETIZER;
```

Appendix XIII

The following assumes the existence of an Ada compiler following UNIX™ naming conventions and that base of a file name (the part before the ‘‘.’’) is the same as the name of the contained module.

```
kwic: kwic.ada input.o ALPHABETIZER.o output.o
        ada -o kwic kwic.ada input.o ALPHABETIZER.o output.o
input.o: input.ada LINE_STORAGE.o
        ada -o input.o input.ada LINE_STORAGE.o
output.o: output.ada ALPHABETIZER.o CIRCULAR_SHIFTS.o LINE_STORAGE.o
        ada -c output.o output.ada ALPHABETIZER.o CIRCULAR_SHIFTS.o \
                LINE_STORAGE.o
ALPHABETIZER.o: ALPHABETIZER.ada CIRCULAR_SHIFTS.o
        ada -o ALPHABETIZER.o ALPHABETIZER.ada CIRCULAR_SHIFTS.o
CIRCULAR_SHIFTS.o: CIRCULAR_SHIFTS.ada LINE_STORAGE.o
        ada -o CIRCULAR_SHIFTS.o CIRCULAR_SHIFTS.ada LINE_STORAGE.o
LINE_STORAGE.o: LINE_STORAGE.ada
        ada -o LINE_STORAGE.o LINE_STORAGE.ada
```

™ UNIX is a trademark of Bell Telephone Laboratories.