Reprinted from:

Journal of Systems and Software
Volume 13
1990
Pages 209–230

The Use of a Repeated Phrase Finder in Requirements Extraction*

Christine Aguilera
Computer Science Department, University of California, Los Angeles, California

Daniel M. Berry
Computer Science Department, Technion, Haifa 32000, Israel

A program, **findphrases**, for finding repeated phrases in an arbitrary text is presented. Its primary intended application is finding the abstractions in problem descriptions. It is hoped that this tool can be used as the basis for an environment to help organize the sentences and phrases of a natural language problem description to aid the requirements analyst in the extraction of requirements. Four experiments to confirm its effectiveness are described. These experiments show that the same abstractions are found by the tool in the natural language description, the final executable program, and in various versions in between.

1. INTRODUCTION

The first steps in the development of any computational system should be the writing of requirements with the client's help. It may be necessary to build a prototype first, but ultimately before building a production-quality version, it is necessary to agree upon what is to be in the system. As observed by Winchester and Estrin [32], well-formed requirements meet a number of requirements themselves.

1. The requirements must be understandable to both the customers and the designers and builders.
2. The parts of the requirements must be consistent with each other.
3. The requirements must be complete so that the designers and builders do not have to make unintended value judgments during their work.
4. Each part of the requirements must be traceable in the subsequent design and implementation.
5. It must be testable that the implementation meets the requirements.
6. The requirements should allow as much design freedom as possible by not specifying more detail than is necessary.

This paper ultimately describes and determines the effectiveness of one tool designed to assist in one part of the process of writing requirements. However, as several referees have pointed out, it is essential that the reader understand the context in which this tool is expected to operate. Hence, Sections 2 through 5 are devoted to describing this context.

2. THE PROBLEM

Many system design or programming methods, e.g., those of Jackson [18], Parnas [24], Booch [10], Myers [21], Orr [22, 23], etc., start from a clear statement of the requirements and show how to arrive at a design of a program or even at a program meeting these requirements. However, none of these methods really explain how these requirements are obtained in the first place. It seems clear to us (at least) that the writing of the requirements is a *major* part of the problem, and that once these are available, the arrival at an implementation, by comparison, is relatively straightforward.

Large E type [19] software, for which it is difficult or even impossible to obtain clear requirements, is usually

Address correspondence to Daniel M. Berry, Computer Science Department, Technion, Haifa 32000, Israel.

* This work was supported in part by the University of California MICRO program, Unisys Corporation, and NCR Corporation.

developed for a client organization in which there are many people who have some view or say as to what the desired system should do. These views range from being totally unrelated to each other to being totally inconsistent with each other. It is no wonder that the distillation of these views into a consistent, complete, and unambiguous statement of the requirements, albeit in natural language, is a *major* part of the problem of developing software which meets the client's needs. Therefore, it is essential to have methods and tools that help in distilling these many views into coherent requirements.

3. PAST WORK

There are already a variety of systems, tools, and methods for dealing with requirements. These include SADT [26, 25], IORL [28], PSL/PSA [30], RDL [32], RSL [5, 6, 7], RML [11] and Burstin's prototype [12] tool. The first two are graphically oriented, and the second of these is automated. The remainder work from highly constrained subsets of English consisting of sentences, each of which states one requirement to which the final implementation must adhere. Those which are automated have tools for working with the sentences and abstractions of the requirements document once these sentences and abstractions have been recognized and stated.

Structured Design and Analysis Technique (SADT) uses a graphic language for expressing the requirements of the system under development. An SADT model is an organized sequence of diagrams starting with a top-level overview diagram. A diagram is composed with at most six boxes connected in an arbitrary manner with arrows and accompanied with explanatory text. Each lower-level diagram is an expansion of one of the boxes at its parent diagram. An arrow between boxes represents a constraint relation between the boxes and not necessarily a flow of control or data.

Teledyne Brown Engineering has been experimenting with a Technology for the Automated Generation of Systems (TAGS), a system development method that has the developers focusing on writing requirements rather than on coding. TAGS is composed of the Input/Output Requirements Language (IORL), a tool system, and the TAGS method. IORL is a graphic and tabular language that allows identification of each important software, hardware, embedded, or management component of the system under development. The system must be specified as a hierarchical collection of components that interact through data links combined with a controlling mechanism that dictates how information flows through the system. The highest level in an IORL specification is a schematic block diagram (SBD) showing the major system components and the data interfacing between them. Each such component may be expanded in a lower level SBD. Associated with each SBD are a variety of other diagrams supplying other information about the components in the SBD. These include diagrams for specifying control, logic, and data flow of and among the individual components of an SBD. The multiview orientation of TAGS is similar to that of SARA [15, 16], except that the latter is directed at implementation design while the former is directed at specification. (The marked similarity of notations for specification and design underscores the fuzziness of the boundary between specification and design.) The associated tool set allows construction of the diagrams, a number of static checks within and between diagrams, and simulations driven by the diagrams.

Problem Statement Language (PSL) is a language for expressing the objects and relations among objects of the system under development. A system consists of things called objects which may have properties which in turn may have values. The objects may be connected by relationships. The language has a rich collection of standard object, property, and relation types that can be used to describe all aspects of an information system, including input, output, data flow, system and data structure, performance, and management. The PSL processor, Problem Statement Analyzer (PSA) builds a relational data base capturing the contents of a PSL specification and prepares a variety of reports. The data base can be used as a living specification that can be interrogated and updated as necessary.

Some of the other requirement statement languages and accompanying tools have adopted PSL/PSA's relational data base orientation. TRW's Software Requirements Engineering Program (SREP) project has produced the Software Requirements Engineering Methodology (SREM) which makes use of a number of tools comprising the Requirement Engineering and Validation System (REVS). REVS is used to translate sentences of Requirement Statement Language (RSL) into tuples of a relational data base called the abstract system semantic model. Tools are provided for interrogating and updating this data base as well as for preparing reports. RSL differs from PSL in that the former emphasizes flows.

Winchester's Requirement Definition Language (RDL) of UCLA's SARA System also expresses relations between objects of the system under development. Each sentence in RDL represents a tuple of a relational

data base built up about the system under development. Some of these relations can be expressed pictorially through the use of SARA's other modeling languages for exhibiting system structure and control and data flows.

The designers of RML observe that requirements are meaningful only in the context of certain real-world knowledge. For example, when the requirements specify a temperature and a tolerance, knowledge about heat, temperature, measurement, tolerance limits, Fahrenheit, Celsius, etc., is implied on the part of the reader and is often used implicilty by the programmers. Thus, specification processing tools should have access to a knowledge base that can be consulted to provide implied details that are not given explicitly. A RML specification organizes the world as a collection of interacting objects. For each object, its characteristics are given as incomplete sentences with the object implied as the missing parameter. Thus an object's specification can be viewed as a collection of relation tuples with the object appearing at least once in each tuple. A complete specification can thus be organized as a relational data base. It is intended to build tools for translating a RML specification into an AI language for knowledge representation so that existing tools can be used for extracting the implied information of a specification.

Burstin's prototype tool allows tuples of a relation, i.e., sentences, each with a verb and objects, to be organized into a hierarchy of abstractions. Each abstraction contains those sentences sharing a common collection of objects, with the verbs representing procedures of the abstraction. There are tools for introducing and moving sentences to and from abstractions and for placing and moving abstractions in the hierarchy. There is also a rudimentary application-oriented expert system that helps recognize when two or more phrases of sentences may be talking about the same thing, e.g., "plane" and "airplane" or "passenger" and "flier."

All of these systems are useful for working with sentences and abstractions of a requirements document, once they are recognized and formed. Organizations implementing and using two of these, PSL/PSA and SREM, report much user satisfaction [7, 27, 30].

It is interesting that all of the above requirements analysis systems deal with relations and that all but the first two, which are not picture-oriented, have gone to the use of a relational data base for storing the relations. All can be used to support abstraction-based requirement development, which leads naturally to abstraction-based software development [9].

However, none of the methods and tools give much help in actually obtaining the sentences in the first place and in recognizing the relevant abstractions, especially in the context of a large client organization. The descriptions of all of the methods either fail to mention how to get the sentences or say something to the effect of "get them and write them down" as if there were nothing to it.

Teichroew and Hershey [30] offer that "since most of the data must be obtained through personal contact, interviews will still be required." PSA does help this gathering process in that its "intermediate outputs ... also provide convenient checklists for deciding what additional information is needed and for recording it for input."

Alford [5] says that the "SREM steps address the sequence of activities and usage of RSL and REVS to generate and validate the requirements. It *assumes* [italics are not in the original] that system function and performances have been allocated to the data processor, and have been collected into a Data Processing Subsystem Performance Requirement or DPSPR."

Even eight years later, Scheffer et al. [27], from a completely different company which had been using SREM, state only that the "initial input to SREM is a system specification that is translated into RSL and interpreted to determine the interfaces with the outside world, the messages across these interfaces, and the required processing relationships and flows."

The first step of the TAGS method [28] is the conceptualization step, "User concepts and requirements are used to develop a conceptual model that is the basis for subsequent engineering." This conceptual model is the top level SBD. In the cited article, there is advice on the issues that should be dealt with in arriving at it. However, no tools are provided, since the TAGS method deals with activities that follow the production of this first SBD.

Therefore, we feel that the gap between the initial fuzzy natural language statements from the individuals in the client organization to the sentences, i.e., relations, with which these tools work is still too large. Methods and tools are needed to close this gap.

4. ENVISIONED REQUIREMENTS GATHERING ENVIRONMENT

We ultimately envision an integrated environment, REGEE, for gathering, sifting, and writing requirements. This environment may very well be part of a larger environment used for software development, deployment, and maintenance [1, 2]. REGEE is in the very rudimentary prototyping stage, as we do not understand the process it is supposed to assist. For now, RE-

GEE is described as helping the human requirements analyst (RA) massage transcripts of interviews with members of a client organization into a consistent, complete, unambiguous, coherent, and concise statement of what the organization wants. We do not care what language is being used either for the interview transcripts or for the final requirements. REGEE should support any possibility. Usually the input, transcript language will be some natural language, possibly with pictures. However, the output language, in which the requirements are written, can be anything from natural language, possibly with pictures, to predicate calculus; in particular, it should be possible to use any of the requirements expression languages that are mentioned in Section 3.

We do not know enough about effective requirements writing to be able to codify the process. Thus at least for now, an expert-system approach is out of the question. We therefore envision an environment consisting of clerical tools that help with the tedious, error-prone steps of what one particular human RA, the second author, does.

We view a requirements document both in the process of being written and in final form as a network, very often a hierarchy, of nodes each denoting an abstraction and containing a description of all that is known and required about the abstraction. The arcs between the nodes can be used to describe the "uses" relation or any other basis for organizing the set of abstractions.

REGEE needs two basic kinds of tools,

1. to help identify the abstractions that will make the nodes from the transcripts of the interviews, and
2. to help organize the abstractions into a network of abstraction-nodes, each to contain a consistent, complete, unambiguous, coherent, and concise description of that abstraction.

This paper deals with a proposed tool of the first kind, a repeated phrase finder. To understand the rationale behind the repeated phrase finder, it is useful to understand the envisioned tool of the second kind. We are building a significant enhancement of the Burstin tool mentioned in Section 3. This tool provides a medium in which nodes, implemented as windows on a work station screen, can be organized into a network, as suggested by Figure 1. Each window can be made to hold arbitrary text, including text that causes displaying of a picture. Any arbitrary element of the text of any window can be given links connecting the element to any window or to any element, possibly in another window. Figure 2 shows two windows from a description of an airline reservation system.

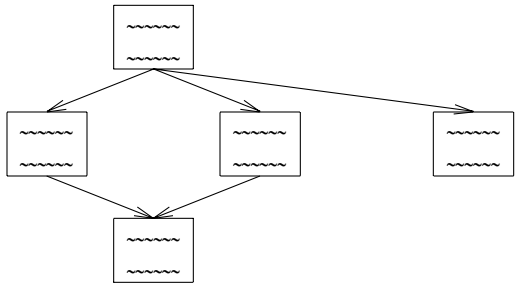

Figure 1. Network of Nodes.

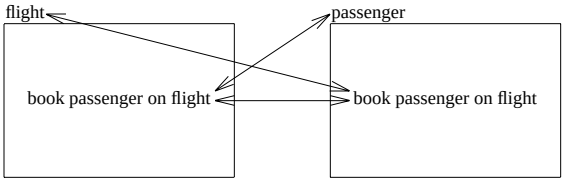

Figure 2. Links.

The links connect an element to windows giving more details about the element or to other elements talking about the same or related concepts, as the human RA desires. The RA can use these links to navigate through the windows as he or she is tracking down the information that allows the contents of each window to be refined into a consistent, complete, unambiguous, coherent, and concise description of the window's abstraction.

This description of the tool of the second kind suggests building it on top of some existing hypertext system [13, 14, 33]. Indeed, Garg and Scacchi have suggested maintaining all life-cycle documents as hypertext [17].

5. ABSTRACTION IDENTIFICATION

The way identification of abstractions is done now is that the human RA scans the transcripts trying to note important subjects and objects of sentences, i.e., nouns. The problem is that humans get tired, get bored, fall asleep, and overlook relevant ideas. So we want a tool that does the clerical part of the search without getting tired, getting bored, falling asleep, and overlooking anything. The human RA still does all the *thinking* with the output of this tool, confident that no occurrence of any noun has been overlooked.

Our first idea, reported in Berry et al. [8], was to use a parser to find the nouns. However, we tried it and found that the few errors it made were so distracting

that it was more comfortable to do it by hand. Furthermore, the program did not inspire confidence that it found everything. Maybe there was an important noun that was overlooked because it appeared to the parser as a verb. Even a better, but still ultimately imperfect, parser does not solve this confidence problem. We want something with guaranteed coverage, even if it is less intelligent. The lack of intelligence in the tool is no problem because a human is applying his or her intelligence to the output of the tool.

The tool we propose for helping the RA to find abstractions is **findphrases**, a repeated phrase finder.

6. FINDING REPEATED PHRASES IN A TEXT

The purpose of the **findphrases** tool is to find repeated phrases in an arbitrary text. **findphrases** is described by its UNIX™-style manual page given in Appendix A.

In its simplest application, the user provides **findphrases** with the text to be analyzed and a file containing punctuation and keywords. The punctuation and keywords are used by **findphrases** to break the text into sentences. **findphrases** processes the phrases of the sentences and produces a series of reports. The basic output contains: (1) the input file as is with lines numbered and the punctuation and keywords overstruck, (2) a frequency ranked table of repeated phrases, and (3) an alphabetically ordered table of repeated phrases. Each entry in these tables gives the numbers of the lines in which the phrase occurs, so that each phrase may be examined in its original context to decide which abstraction is really represented by the phrase. A number of options are provided that the user may use to control the parsing of the input text into tokens and phrases, to control the printing of the phrases in the tables of the output, and to indicate which additional tables are to be printed. Sample outputs from several applications of **findphrases** on a variety of texts are provided in Appendix C. These applications are described in Section 10.

In another software project, Takata [29] has used **findphrases** to help identify the phrases of a book that should be indexed in conjunction with a tool which generates a formatted index from the text of a book and the list of phrases to be indexed. Takata's tool, **indx**, takes a *phrase-file* which contains all phrases for which the index is to be developed.

findphrases may also be used in other applications where the identification of repeated phrases in a text is needed.

Observe that there is a learning process involved in using **findphrases** effectively for each of these and other applications. First, it appears that there are different characteristic punctuation-keyword, multitoken and initial ignored-phrases files for each language. These can be catalogued for general use. In addition for each class of applications, there appears also to be a characteristic set of additional ignored phrases. Finally, as one is doing a particular application, one finds it useful to extend the ignored phrases file with common words that are actually important abstractions, but whose presence skews the list and populates it with too much noise for finding the other abstractions.

7. TESTING EFFECTIVENESS WITH FOUR EXPERIMENTS

All of the above are just raw ideas about how to build a requirements gathering and writing environment and a plausable explanation of why it should work. However, unless the **findphrases** program is effective in helping the human RA to identify abstractions, it is useless to continue down this path. The first author implemented **findphrases** and conducted tests to determine its effectiveness in its intended purpose.

We found four examples of program development, each of which had multiple versions of the same program ranging from natural language descriptions, through designs, decompositions, etc., to code. Three of these are published in the literature and one is not; none were designed to be a subject of this experiment (our own included), although obviously they were *picked* to be subjects after the fact.

It is desired to determine if **findphrases** is effective in helping the human RA to find, in the natural language transcripts of interviews about a system under development, *all* of the abstractions that serve as the basis for the requirements, design, and implementation. It will be deemed effective if we, as humans, do indeed recognize the same set of abstractions in the outputs of **findphrases** run (with the appropriate parameter files in each case) on all versions of the same problem. Finding the same set of abstractions in all versions says that the abstractions found in the first version, the natural language description, are sufficient to cover all abstractions that will be needed for all subsequent versions, including the code, and that no other abstractions will need to be invented.

While doing the experiments, we were discovering methods for finding abstractions and producing the requirements from the initial statements. However, the refinement of these methods is left to future work if and when **findphrases** has been judged effective.

The sections that follow describe the four experimental applications of **findphrases**. The relevant output list-

ings for some of the examples are found in the appendices. Space considerations prevent giving the output listings for all of the examples. However, they may be found in the appendices of Aguilera's thesis [4].

The first experiment is Abbott's example of programming with the help of natural language. This example is the focus of a paper [8] that points to the need of this phrase-finding tool. For this experiment, three versions of the program solution are compared. The first version was written in standard English, the second in an Ada-based program design language, and the third in Ada. Since Berry et al. [8] is published in a previous issue of this journal, and the publication contains ouptuts of some runs of an earlier version of **findphrases** on some of the versions of the problems, the outputs of the runs are not given here.

The second experiment is the problem of writing the phrase finder itself. In writing the phrase finder, the manual page served as the requirements document. **findphrases** was run with its own manual page to see if the same abstractions that formed the basis for the modular decomposition used in writing the code are identified from the information provided by **findphrases**. The manual page itself is given as Appendix A, and the frequency listing is given as Appendix B.

The third experiment takes Mitchell's textbook [20] example of writing a sorting program starting from the English statement of the requirements and ending with a Pascal program developed with a structured programming method. Four versions of the program solution are compared, the initial English description, two program design language descriptions, and the final Pascal program. The input and output from the runs of **findphrases** on all four versions of the problem are given in Appendix C.

The fourth experiment takes Wiener and Sincovec's textbook [31] example of writing a spelling checker program. They start with a statement of the requirements, develop a modular decomposition for the solution, and produce an Ada program. **findphrases** was used with the list of general goals and requirements for the spelling checker. The abstractions found with the aid of the output produced by **findphrases** in this application are compared with those used by Wiener and Sincovec. The listings from these runs are not given in this paper.

8. ABBOTT'S EXAMPLE — THE NUMBER OF DAYS BETWEEN TWO DATES

In this experiment, three versions of a problem and its solution were tested. The problem to be solved was "write a function subprogram that, given two dates in the same year, returns the number of days between the two dates." [3] The first solution was Abbott's original informal strategy written in English, the second was the strategy written in an Ada-based program design language, and the third was the final Ada program. The second and third versions were taken from Berry et al. [8].

findphrases was applied to each version using the appropriate *punctuation-keyword-file* and other options provided by **findphrases**. Appendix B of Aguilera [4] lists the input text, the selected options, and the tables generated by **findphrases** for each version.

To compare the abstractions contained in the different versions of the solution, a method was needed for finding the abstractions from the tables of repeated phrases that are produced by **findphrases**. The method selected was Abbott's own method of formalizing an informal strategy [3].

An informal strategy can be formalized into a program by

1. identifying the data types,
2. identifying the objects (program variables) of those types,
3. identifying the operators to be applied to those objects, and
4. organizing the operators into the control structure suggested by the informal strategy.

This is done by looking at the English words and phrases of the informal strategy and, respective to the list above,

1. identifying the common nouns,
2. identifying the proper nouns or direct references
3. identifying the verbs, attributes, predicates, or descriptive expressions suggesting an operator, and
4. identifying the control structures that are implied in a straightforward way by the natural language description, e.g., by use of phrases such as "if," "then," "otherwise," "for each," etc.

For this experiment, common nouns were used to identify the data abstractions. Using the tables of repeated phrases, the user can manually identify the common nouns since these tend to be repeated very often. To determine whether a noun actually represents a distinct data type, its context must be considered. By using the line numbers printed in the tables, or by using the UNIX **grep**, the user can locate each noun, analyze the context of the lines containing it, and make a determination. It is possible that a common noun appears only once in the text and therefore does not appear in the repeated phrases tables. To handle such cases, the `-t` op-

tion of **findphrases** allows the user to obtain a listing of all the distinct tokens in the text. Using this list, the user can identify missing common nouns that in turn can be located and analyzed. In his discussion of the informal strategy, Abbott identifies "counter" as a common noun, a phrase that only appears once according to **findphrases**. In the text, however, the underscore was used in the phrase `Day_counter` thus capturing the multiple occurrences of "counter." Because the phrase `Day_counter` is included in the repeated phrases tables, the user may note the occurrence of the common noun "counter," or by examining the Tokens List in Appendix B, Section 1 of Aguilera [4], the single occurrence of `counter` can be identified. In this case, after analyzing its usage, this phrase was found to be equivalent to the data type `number_of_days` and was therefore not used as a separate data abstraction.

In the informal strategy, the common nouns identified as actual data types were only `date`, `month`, and `number_of_days`. These same phrases were identified by **findphrases** as repeated phrases in all three versions of the problem description. Other common nouns found in the different versions occurred as equivalent rephrasings of these same types. For instance, in the informal strategy, `dates`, `days`, and `number` also occurred as repeated phrases. Since the second and third versions were developed through an iterative process, the results from **findphrases** showed that once the initial data types were identified, they were used consistently in each of the later versions.

In this experiment, the data types that form the basis of the data abstraction `calendar_tools` were identified as repeated phrases by **findphrases** in each version of the problem description. Thus, **findphrases** was found to be effective in helping the human RA identify the same set of abstractions in each version. In addition, the tables produced by the **findphrases** tool were found to be useful in organizing the information of the text while applying Abbott's method.

9. THE **findphrases** MANUAL PAGE

This experiment considered the problem of developing a program for **findphrases**. **findphrases** was tested using its own manual page. The data abstractions identified by **findphrases** from this application were compared with the abstractions used in the decomposition for the final program. Appendix B contains one of the tables produced by **findphrases** using the manual page.

The decomposition used for the **findphrases** program includes 15 different modules, as illustrated in Figure 3. (See Chapter 3, Section 3 of Aguilera [4] for a description of the modules.) The modules with dashed outlines are built into the implementing programming language. Nine of the non-built-in modules, i.e., those with the thicker outlines, are data abstraction packages. The list below shows the correspondence between these packages and some of the repeated phrases identified by **findphrases** from the manual page description:

The Data Abstraction Packages	**The Repeated Phrases**
`string_type_file`	`strings, character strings`
`argument_line`	`argument, option`
`output_file`	`output, tables of the output`
`chunk_file`	`file(s), free-format`
`punct_keyword_table`	`punctuation, keyword(s), punctuation/keyword(s), punctuation-keyword-file`
`multi_tokens_table`	`multi-tokens, multi-tokens-file`
`text_file`	`text, input, arbitrary text`
`phrases`	`phrase(s), ignored phrases, repeated phrases`
`sentences`	`sentence(s)`

The name of each abstraction, except for `chunk_file`, directly corresponds to at least one repeated phrase appearing in the manual page. Although the phrase "chunk" does not appear in the description, the `chunk_file` abstraction is implied by the phrases indicated above. During the program's development, `chunk_file` was recognized as the abstraction to be used by the procedures that read the various free-format files. Thus, this abstraction is indirectly identified by repeated phrases appearing in the tables.

In this experiment, the repeated phrases tables produced by **findphrases** include phrases that identify the abstractions that were used in the program decomposition. The tables act as guides to the user when looking for the abstractions. The user's attention is first focused on the phrases with the highest frequency and then on possibly related phrases. The line numbers printed in the tables enable the user to locate the phrases in the text. By analyzing the context of the sentences containing the phrase, the importance of the phrase as a potential abstraction can be determined.

In addition to finding the list of abstractions, as the user analyzes the context of each phrase usage, a list of statements related to that phrase can be extracted. For example, consider the phrases `multi-token`, `multi-tokens` and `multi-tokens-file`. By exa-

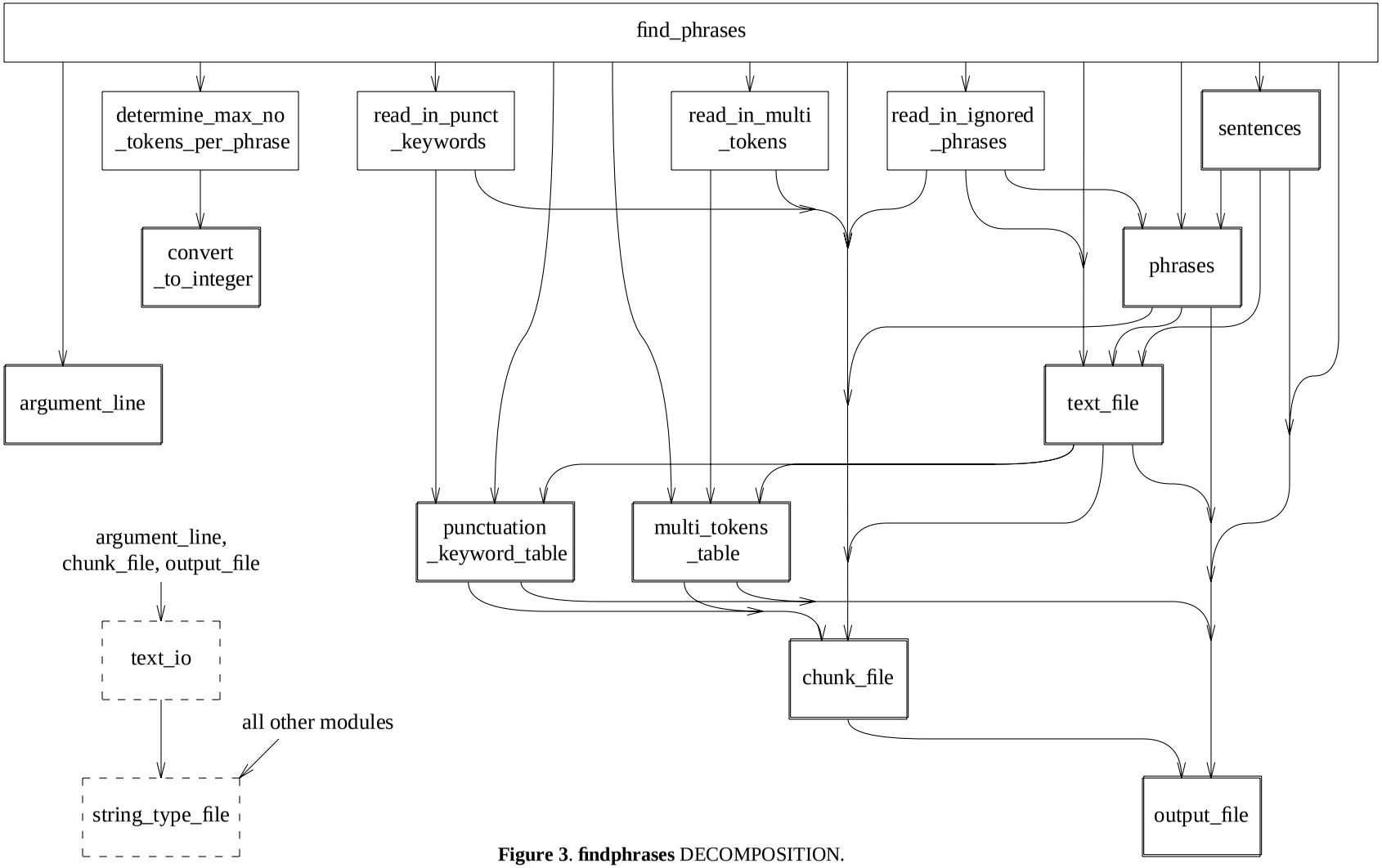

Figure 3. **findphrases** DECOMPOSITION.

mining the context of the lines that contain these phrases, one finds the following information:

1. The optional *multi-tokens-file* contains in free format the list of character strings to be taken as multitokens;
2. a multitoken is a string consisting of more than one symbolcharacter (non-word character);
3. if the `–m` option is present, the input text is parsed using the information provided by the *multi-tokens-file*;
4. a token may be a multitoken;
5. if the `–v` and the `–m` options are present, the list of multitokens is printed.

These statements represent requirements for the data abstraction `multi-token`. Located by the manual method, these same statements were used when the initial decomposition and implementation were developed. The final program uses a procedure to read the *multi-tokens-file* and a package to handle a multitokens table. These statements also guided the development of the text parsing routines.

Thus, in this experiment, the human RA finds in the repeated phrases generated from the manual page the very abstractions used in the modular decomposition and implementation.

10. AN ALGORITHM FOR SORTING A LIST OF NAMES

The third experiment used the problem of sorting a list of names alphabetically. The problem was taken from Mitchell [20] from the chapter titled "Describing Algorithms in English."

From Mitchell's algorithm, a fully implemented Pascal program was developed using three iterations of Abbott's method of formalizing an informal strategy. **findphrases** was tested with all four versions: (1) the initial English description, (2) a program design language description using **`goto`** statements, (3) a program design language description with the **`goto`** statements eliminated, and (4) the final Pascal program. Appendix C lists the results from applying **findphrases** to each version.

Using Abbott's initial step of identifying the common nouns in the tables produced by **findphrases** for the informal strategy, the data types `list`, `name`, and `position` were identified. These phrases, appearing in equivalent forms, occurred as repeated phrases in all four versions of the problem description and were used as abstractions in the final program. Appendix C contains the inputs and outputs of the runs of **findphrases** on these four versions.

For this experiment, **findphrases** was found effective in helping to identify the abstractions used in the final program from each version representing various stages in the development of the problem solution.

11. WIENER AND SINCOVEC'S SPELLING CHECKER

This final example comes from *Software Engineering with Modula-2 and Ada* by Wiener and Sincovec [31]. The chapter titled "A Case Study in Modular Software Constructions" lists the goals and requirements of the spelling checker and describes an informal strategy along with a design framework and implementation. **findphrases** was tested with a text that describes the goals and requirements of the spelling checker. The output from this application is listed in Appendix E of Aguilera [4].

In the spelling checker case study, Wiener and Sincovec give the modular design used to implement the spelling checker. The modules include the main program `Spell`, the procedure `TestWord`, and seven data abstractions: `Hidden`, `TextOps`, `Counters`, `MainDict`, `FastDict`, `AuxDict`, and `TempDict`.

Using the tables produced by **findphrases**, the potential data types were identified using methods similar to those used in the previous experiments. The repeated phrases identified as data types were: `word`, `line`, `text file`, `dictionary`, and `number of words`. From the context of the sentences containing the phrase `line`, the data types `line of text` and `number of lines` were implied.

The data abstractions found compare favorably with those used by Wiener and Sincovec. In their implementation, Wiener and Sincovec used `Hidden` to hide the data types `WordType` and `LineType` that correspond to `word` and `line of text`. Their module `TextOps` was used to handle the `text file`. The four dictionary modules, `MainDict`, `FastDict`, `AuxDict`, and `TempDict`, were used to handle the dictionaries that correspond to the data type `dictionary`. A separate module called `Counters` was used to handle the operations on the counters for the `number of words` and `number of lines`. These comparisons show that the data abstractions found by using **findphrases** correspond directly to those used by Wiener and Sincovec in their implementation.

As with the previous experiments, the data abstractions identified from the repeated phrases in the initial goals and requirements description of the spelling checker are exactly those used in the final implementation.

12. TEST RESULTS AND CONCLUSIONS

In these experiments, **findphrases** was found to be effective in helping the human RA identify the same abstractions from different versions of a program description. In addition, **findphrases** was helpful in organizing the information in the descriptions and allowed the user to focus on the important phrases. It will now be necessary to try out **findphrases** on new and larger problems.

In addition, methodological work is needed. Specifically, initial punctuation-keyword, multitoken, and ignored phrases files need to be suggested for each language and problem area. Strategies of adding words to the ignored phrases file must be developed. More work is needed to consider the appropriate methods for reducing the list of repeated phrases down to the list of abstractions.

Finally, it is necessary to build the hypertext-based requirements massager and to complete the construction of REGEE.

ACKNOWLEDGEMENT

The authors thank Yoëlle Maarek and the referees for their useful comments. UNIX is a trademark of AT&T Bell Laboratories.

REFERENCES

1. Requirements for the Ada programming support environment: STONEMAN, Technical Report, U.S. Department of Defense (1981).
2. *IEEE Software*, 5 (1988).
3. R.J. Abbott, Program design by informal english descriptions, *CACM*, 26 (1983).
4. C.S. Aguilera, Finding abstractions in problem descriptions using **findphrases**, M.S. Thesis, Computer Science Department, UCLA, Los Angeles (1987).
5. M.W. Alford, A requirements engineering methodology for realtime processing requirements, *IEEE Transactions of Software Engineering*, SE-3, 60–69 (1977).
6. M.W. Alford, Software requirements engineering methodology (SREM) at the age of two, in *COMPSAC 78 Proceedings* (1978).
7. M.W. Alford, SREM at the age of eight; the distributed computing design system, *Computer*, 18, 36–46 (1985).
8. D.M. Berry, N.M. Yavne, and M. Yavne, Application of program design language tools to abbott's method of program design by informal natural language descriptions, *Journal of Software and Systems*, 7, 221–247 (1987).
9. V. Berzins, M. Gray, and D. Naumann, Abstraction-based software development, *Communications of the ACM*, 29, 402–415 (1986).
10. G. Booch, *Software engineering with Ada*, Benjamin-Cummins, San Francisco, 1986.
11. A. Borgida, S. Greenspan, and J. Mylopolous, Knowledge representation as the basis for requirements specifications, *Computer*, 18, 82–91 (1985).
12. M.D. Burstin, Requirements analysis of large software systems, Ph.D. Dissertation, Department of Management, Tel Aviv University, Tel Aviv, Israel (1984).
13. J. Conklin, A survey of hypertext, MCC Technical Report No. STP-356-86, Rev. 1, MCC, Austin, TX (1987).
14. N.M. Delisle and M.D. Schwartz, Contexts — a partitioning concept for hypertext, *ACM Transactions on Office Information Systems*, 5, 168–186 (1987).
15. G. Estrin, The story of SARA, *Proceedings of IFIP Working Conference on Methodology for Computer System Design* (1983).
16. G. Estrin, R.S Fenchel, R.R. Razouk, and M.K. Vernon, SARA (System ARchitect's Apprentice): modeling, analysis, and simulation support for design of concurrent systems, *IEEE Transactions of Software Engineering*, SE-12, 293–311 (1986).
17. P.K. Garg and W. Scacchi, Maintaining software life cycle documents as hypertext: issues, analysis, and directions, Technical Report, University of Southern California, Los Angeles (1987).
18. M.A. Jackson, *Principles of program design*, Academic Press, London, 1975.
19. M.M. Lehman, Programs, life cycles, and laws of software evolution, *Proceedings of the IEEE*, 68, 1060–1076 (1980).
20. W. Mitchell, *A prelude to programming: problem solving and algorithms*, Reston Publishing, Reston, VA, 1984.
21. G.J. Myers, *Composite/structured design*, van Nostrand Reinhold, New York, 1979.
22. K.T. Orr, *Structured systems development*, Yourdon, New York, 1977.
23. K.T. Orr, *Structured requirements engineering*, Ken Orr & Associates, Topeka, KS, 1981.
24. D.L. Parnas, On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, 15, 1053–1058 (1972).
25. D.T. Ross, *IEEE Transactions on Software Engineering*, SE-3, 16–33 (1977).
26. D.T. Ross and K.E. Schoman, Jr., Structured analysis for requirements definition, *IEEE Transactions on Software Engineering*, SE-3, 6–15 (1977).
27. P.A. Scheffer, A.H. Stone, III, and W.E. Rzepka, A case study of SREM, *Computer*, 18, 47–54 (1985).
28. G.E. Sievert and T.A. Mizell, Specification-based software engineering with TAGS, *Computer*, 18, 56–66 (1985).
29. K.K. Takata, Indx, a semi-automatic indexing program, M.S. Thesis, Computer Science Department, UCLA, Los Angeles (1987).

30. D. Teichroew and E.A. Hershey, III, PSL/PSA: a computer-aided technique for structure documentation and analysis of information processing systems, *IEEE Transactions of Software Engineering*, SE-3, 41–48 (1977).
31. R. Wiener and R. Sincovec, *Software engineering with Modula-2 and Ada*, John Wiley & Sons, New York, 1984.
32. J. Winchester and G. Estrin, Requirements definition and its interface to the SARA design methodology for computer-based systems, *AFIPS Conference Proceedings*, 51, 369–379 (1982).
33. N. Yankelovich, N. Meyerowitz, and A. van Dam, Reading and writing the electronic book, *Computer*, 10, 15–30 (1985).

APPENDIX A. **findphrases MANUAL PAGE**

FINDPHRASES (LOCAL) UNIX Programmer's Manual FINDPHRASES (LOCAL)

NAME

findphrases – find repeated phrases in an arbitrary text

SYNOPSIS

findphrases [**–n***number*] **–p***punctuation-keyword-file* [**–x***ignored-phrases-file*] [**–m***multi-tokens-file*] [**–u**] [**–b**] [**–s**] [**–t**] [**–v**] [**–c**]

DESCRIPTION

All files mentioned in the synopsis provide their data in what is referred to as free format subject to particular restrictions to be described for each case. In free format, the items of the file may be entered zero or several per line with a mixture of blanks and tabs before, in between, and after the items. Consequently, no item can include a blank, a tab, or a newline.

The **–n** argument is optional and if present provides a number *number* serving as the maximum length phrase (to be described later) to be tallied. If this argument is not present, if it does not supply a number, or if the supplied number is outside the reasonable range of greater than zero and less than or equal to 50, then *number* is taken as 10.

The *punctuation-keyword-file* contains in free format a list of those character strings to be taken as punctuation/keywords (see below). The optional *ignored-phrases-file* contains one-per-line a list of phrases to be ignored in the tallying (see below). In each line, the tokens (see below) are in free format. The optional *multi-tokens-file* contains in free format a list of those character strings consisting of more than one symbolcharacter (see below) which are to be taken as multi-tokens (see below).

No assumptions are made about the standard input, thus it may be an arbitrary text. The program parses the text into words and symbolcharacters. These in turn are formed and classified into tokens and punctuation/keywords based on the information provided by the *punctuation-keyword-file* and, when the **–m** option is present, the *multi-tokens-file*.

First some definitions are necessary:

Whitespace: blank, *tab*, *newline*, *beginning-of-file*, *end-of-file*

Wordcharacter: letter, *digit*, _

Symbolcharacter: any printable character which is neither a wordcharacter nor a blank

Word: any sequence of wordcharacters delimited on each side by whitespace or a symbolcharacter

Punctuation/Keyword: whatever is in the *punctuation-keyword-file*; the symbolcharacter strings are called punctuation and the wordcharacter strings are called keywords

Multi-token: whatever is in the *multi-tokens-file*

Token: any word, symbolcharacter, or multi-token which is not listed in the *punctuation-keyword-file*

Sentence: list of tokens delimited on each side by punctuation/keyword

Phrase: one or more consecutive tokens occurring within one sentence

The main job of this program is to tally the occurrence of all phrases in all sentences. The maximum length phrase that has to be considered is that of *number* tokens. If the *ignored-phrases-file* is provided, then the phrases given in the file are to be ignored in the tallying. If the **–b** option is used along with the *ignored-phrases-file*, then phrases which begin with an ignored phrase are also ignored in the tallying.

The standard output consists of:

a copy of the input as is, with the lines numbered and the punctuation/keywords overstruck two times (i.e., printed three times in place) so that they can be spotted easily,

a frequency ranked table of the repeated phrases. i.e., those appearing more than once among the sentences; that is the entries of the table are given in order of decreasing frequency, and

an alphabetically ordered table of the repeated phrases.

In the two tables, the entry for a repeated phrase consists of:

a sequence of asterisks indicating the phrase's frequency as a percentage of the maximum frequency; in this one asterisk represents 10%,

the actual number of occurrences of the repeated phrase,

the repeated phrase itself, and

a list of the numbers of all lines containing the beginning of the repeated phrase.

In printing the repeated phrase itself in a table entry, the underscores, i.e., ''_'', are printed as blanks. This means that an underscore can be used immediately preceeding or following a word that looks like a keyword to prevent it from being considered a keyword.

Note that the definition of ''phrase'' is independent of the number of times it occurs in the sentences. An *ignored phrase* is simply one to be ignored in the tallying but not in searching for phrases. A phrase which contains an ignored phrase which itself is not ignored is to be tallied. When the **−b** option is present, a phrase which begins with an ignored phrase is not to be tallied. A *repeated phrase* is one whose final tally is greater than one. Only the repeated phrases show up in the tables of the output.

Typically, the *ignored-phrases-file* will contain so-called noise phrases such as ''a'', ''an'', ''the'', ''of'', ''of the'', etc. plus any useless phrases found in previous runs of the program.

One particular configuration of the files is as follows:

Punctuation-keyword-file: **; [ ] abort accept access all and array at begin body case constant declare delta digits do else elsif end entry exception exit for function generic goto if in is limited loop mod new not null of or others out package pragma private procedure raise range record rem renames return reverse select separate subtype task terminate then type use when while with xor**

Multi-tokens-file: **** := <= >= /= .. <> << >>**

This configuration is suited for finding repeated phrases in Ada or in an Ada-based program design language.

If the **−u** option is present, then only the unique phrases that are not wholly and everywhere contained in another phrase are listed in the tables of the output. In addition to the already specified output, if the **−s** option is present, then all the sentences are listed; if the **−t** option is present, then all the tokens are listed; if the **−v** option is present, then the output is verbose with the punctuation/keywords listed, and when the **−m**, and respectively the **−x**, option is present, the multi-tokens, and respectively the ignored phrases, are listed. If the **−c** option is present, then upper and lower case distinctions are to be applied in determining whether a phrase is in a sentence. The default is to ignore case distinction in the comparisons.

DIAGNOSTICS

They are good, of course.

BUGS

There are none, of course.

APPENDIX B. **findphrases** USED ON ITS OWN MANUAL PAGE

The outputs referred to in these appendices follow the entire text. In each case, the output, given in a typewriter font and extending over several pages, is headed by a label including the name of the example. This label is in Roman fonts. The outputs have been edited for compactness and lines have occasionally been folded to fit the column width.

The original text of the manual page is found in Appendix A. The text was modified slightly by the addition of some punctuation before it was used with the tool. The following **findphrases** options were used:

i. The *punctuation-keyword-file* consisted of a standard set of punctuation: period, comma, colon, semi-colon, question mark, and exclamation point.
ii. The *ignored-phrases-file* consisted of a list of sixty-

seven phrases to be ignored: apostrophe, opening and closing double quotes, opening and closing parentheses, opening and closing brackets, dash, colon, underscore, `10`, `a`, `ada`, `all`, `an`, `and`, `any`, `are`, `as`, `based`, `be`, `begin`, `beginning`, `below`, `by`, `called`, `can`, `configuration`, `course`, `described`, `e`, `each`, `end`, `entry`, `file`, `for`, `i`, `if`, `in`, `into`, `is`, `it`, `items`, `may`, `not`, `number is`, `of`, `on`, `or`, `respectively`, `see`, `so`, `synopsis`, `taken`, `than`, `that`, `the`, `then`, `this`, `those`, `thus`, `times`, `to`, `tokens`, `(see below)`, `when`, `which`, and `with`. These phrases were found in prior runs.

iii. The *multi-tokens-file* consisted of the following symbols: `''` `‘‘` `**` `:=` `<=` `>=` `/=` `..` `<>` `<<` `>>`

iv. The `–u` and `–b` options were used to print the Tables of Repeated Phrases. The `–b` option was used to ignore in the tallying of repeated phrases those phrases which began with an ignored phrase. The number used with the `–n` option was `11`.

The output for this experiment, labelled “**Output 1**” shows the text subjected to **findphrases** and the table of repeated phrases sorted by frequency. The alphabetically sorted table of repeated phrases has been omitted.

APPENDIX C. A SORTING ALGORITHM

This appendix contains the results of using **findphrases** on four versions of a sorting algorithm. The first version of the sorting algorithm was taken from [Mit84]. The second through fourth versions were developed using Abbott’s method. In the second version the **`goto`** statements of the first version remained. In the third version the **`goto`**s were eliminated and replaced with **`while`** loops. The fourth version is part of a fully implemented Pascal program. The outputs for these experiments are labelled “**Output 2**” through “**Output 5**.” They each contain the text subjected to **findphrases** and only the frequency ordered table of repeated phrases.

C.1. VERSION 1: AN INFORMAL STRATEGY FOR SORTING

The input file used in the first version of the Sorting Example is listed below as the Input Textfile. The lines of the text have been numbered. The following **findphrases** options were used:

i. The *punctuation-keyword-file* consisted of a comma and period.

ii. The *ignored-phrases-file* consisted of `and`, `be`, `in`, `in the`, `of`, `of the`, `the`, and `to` as phrases to be ignored.

iii. There were no multi-tokens used with this input file.

iv. The `–u` option was used to print the table. The number used with the `–n` option was `5`.

C.2. VERSION 2: THE SORTING STRATEGY WITH **`goto`**S

The input file used in the second version of the Sorting Example is listed below as the Input Textfile. The lines of the text have been numbered and the punctuation and keywords have been highlighted. The following **findphrases** options were used:

i. The *punctuation-keyword-file* consisted of the period, colon, and semi-colon as punctuation and `Algorithm`, `do`, `else`, `end`, `if`, `then`, and `while` as keywords.

ii. The *ignored-phrases-file* consisted of list of nineteen phrases to be ignored. These phrases were found on previous runs.

iii. The *multi-tokens-file* consisted of the symbol `<--`

iv. The `–u` option was used to print the table. The number used with the `–n` option was `16`.

C.3. VERSION 3: THE SORTING STRATEGY WITH **`goto`**S REMOVED

The input file used in the third version of the Sorting Example is listed below as the Input Textfile. The lines of the text have been numbered and the punctuation and keywords have been highlighted. The following **findphrases** options were used:

i. The *punctuation-keyword-file* was the same as the one used with the second version.

ii. The *ignored-phrases-file* was the same as the one used with the second version.

iii. The *multi-tokens-file* consisted of the assignment symbol `:=`

iv. The `–u` option was used to print the table. The number used with the `–n` option was `8`.

C.4. VERSION 4: A PASCAL IMPLEMENTATION FOR THE SORTING STRATEGY

The input file used in the fourth version of the Sorting Example is listed below as the Input Textfile. The lines of the text have been numbered and the punctuation and keywords have been highlighted. The following **findphrases** options were used:

i. The *punctuation-keyword-file* consisted of the period, semi-colon, and opening and closing curly brackets as single symbol punctuation, Pascal opening and closing comment symbols, `(*` and `*)`, as multi-symbolcharacter punctuation, and `begin`, `do`, `end`, `if`, `then`, `while` as keywords.
ii. The *ignored-phrases-file* consisted of the opening and closing parentheses and comma as phrases to be ignored.
iii. The *multi-tokens-file* consisted of the assignment symbol `:=`
iv. The `–u` option was used to print the table. The number used with the `–n` option was `5`.

Output 1. Manual Page.

```
          THE INPUT TEXTFILE

 1  FINDPHRASES(1)                                    UNIX Programmer's Manual.
 2
 3  NAME:
 4       findphrases - find repeated phrases in an arbitrary text.
 5
 6  SYNOPSIS:
 7       findphrases [ -nnumber ] -ppunctuation-keyword-file [ -xignored-phrases-file ]
 8                   [ -mmulti-tokens-file ] [ -u ] [ -b ] [ -s ] [ -t ] [ -v ] [ -c ].
 9
10  DESCRIPTION:
11
12   All files mentioned in the synopsis provide their data in what is referred to as free format
13   subject to particular restrictions to be described for each case. In free format, the items of the
14   file may be entered zero or several per line with a mixture of blanks and tabs before, in
15   between, and after the items. Obviously, no item can include a blank, a tab, or a newline.
16
17   The -n argument is optional and if present provides a number number serving as the
18   maximum length phrase (to be described later) to be tallied. If this argument is not present, if
19   it does not supply a number, or if the supplied number is outside the reasonable range of
20   greater than zero and less than or equal to 50, then number is taken as 10.
21
22   The punctuation-keyword-file contains in free format a list of those character strings to be taken
23   as punctuation/keywords (see below). The optional ignored-phrases-file contains one-per-line a
24   list of phrases to be ignored in the tallying (see below). In each line, the tokens (see below)
25   are in free format. The optional multi-tokens-file contains in free format a list of those character
26   strings consisting of more than one symbolcharacter (see below) which are to be taken as
27   multi-tokens (see below).
28
29   No assumptions are made about the standard input, thus it may be an arbitrary text. The
30   program parses the text into words and symbolcharacters. These in turn are formed and
31   classified into tokens and punctuation/keywords based on the information provided by the
32   punctuation-keyword-file and, when the -m option is present, the multi-tokens-file.
33
34   First some definitions are necessary:
35
36    Whitespace: blank, tab, newline, beginning-of-file, end-of-file.
37
38    Wordcharacter: letter, digit,  _  .
39
40    Symbolcharacter: any printable character which is neither a wordcharacter nor a blank.
41
42    Word: any sequence of wordcharacters delimited on each side by whitespace or a
43   symbolcharacter.
44
45    Punctuation/Keyword: whatever is in the punctuation-keyword-file; the symbolcharacter
46   strings are called punctuation and the wordcharacter strings are called keywords.
47
48    Multi-token: whatever is in the multi-tokens-file.
49
50    Token: any word, symbolcharacter, or multi-token which is not listed in the
51   punctuation-keyword-file.
52
53    Sentence: list of tokens delimited on each side by punctuation/keyword.
```

Phrase: one or more consecutive tokens occurring within one sentence.

The main job of this program is to tally the occurrence of all phrases in all sentences. The maximum length phrase that has to be considered is that of number tokens. If the ignored-phrases-file is provided, then the phrases given in the file are to be ignored in the tallying. If the -b option is used along with the ignored-phrases-file, then phrases which begin with an ignored phrase are also ignored in the tallying.

The standard output consists of:

a copy of the input as is, with the lines numbered and the punctuation/keywords overstruck two times (i.e. printed three times in place) so that they can be spotted easily,

a frequency ranked table of the repeated phrases. i.e. those appearing more than once among the sentences; that is the entries of the table are given in order of decreasing frequency, and

an alphabetically ordered table of the repeated phrases.

In the two tables, the entry for a repeated phrase consists of:

a sequence of asterisks indicating the phrase's frequency as a percentage of the maximum frequency; in this one asterisk represents 10%,

the actual number of occurrences of the repeated phrase,

the repeated phrase itself, and

a list of the numbers of all lines containing the beginning of the repeated phrase.

In printing the repeated phrase itself in a table entry, the underscores, i.e., ''_'', are printed as blanks. This means that an underscore can be used immediately preceding or following a word that looks like a keyword to prevent it from being considered a keyword.

Note that the definition of ''phrase'' is independent of the number of times it occurs in the sentences. An ignored phrase is simply one to be ignored in the tallying but not in searching for phrases. A phrase which contains an ignored phrase which itself is not ignored is to be tallied. When the -b option is present, a phrase which begins with an ignored phrase is not to be tallied. A repeated phrase is one whose final tally is greater than one. Only the repeated phrases show up in the tables of the output.

Typically, the ignored-phrases-file will contain so-called noise phrases such as ''a'', ''an'', ''the'', ''of'', ''of the'', etc. plus any useless phrases found in previous runs of the program.

One particular configuration of the files is as follows:

Punctuation-keyword-file: ; [ ] abort accept access all and array at begin body case constant declare delta digits do else elsif end entry exception exit for function generic goto if in is limited loop mod new not null of or others out package pragma private procedure raise range record rem renames return reverse select separate subtype task terminate then type use when while with xor.

Multi-tokens-file: ** := <= >= /= .. <> << >> .

This configuration is suited for finding repeated phrases in Ada (Ada is a trademark of the U. S. Department of Defense.) or in an Ada-based program design language.

If the -u option is present, then only the unique phrases that are not wholly and everywhere contained in another phrase are listed in the tables of the output. In addition to the already specified output, if the -s option is present, then all the sentences are listed; if the -t option is present, then all the tokens are listed; if the -v option is present, then the output is verbose with the punctuation/keywords listed, and when the -m, and respectively the -x, option is present, the multi-tokens, and respectively the ignored phrases, are listed. If the -c option is present, then upper and lower case distinctions are to be applied in determining whether a phrase is in a sentence. The default is to ignore case distinction in the comparisons.

DIAGNOSTICS:
    They are good, of course.

```
124
125  BUGS:
126       There are none, of course.
127
     *** End of the Input TextFile ***
```

THE REPEATED PHRASES TABLE -- SORTED BY FREQUENCY

Relative Frequency Percentage	Phrase Count	The Phrase and the Lines Containing It	
**********	20	file	7, 7, 8, 14, 22, 23, 25, 32, 32, 36, 36, 45, 48, 51, 59, 59, 60, 97, 102, 108
**********	19	phrase	18, 55, 58, 61, 75, 77, 80, 82, 84, 86, 90, 91, 92, 92, 93, 93, 94, 114, 120
**********	19	phrases	4, 7, 23, 24, 57, 59, 59, 60, 60, 69, 73, 92, 94, 97, 97, 98, 110, 113, 118
*******	14	ignored	23, 24, 58, 59, 60, 61, 61, 91, 91, 92, 92, 93, 97, 118
*******	13	tokens	8, 24, 25, 27, 31, 32, 48, 53, 55, 58, 108, 116, 118
******	12	punctuation	22, 23, 31, 32, 45, 45, 46, 51, 53, 65, 102, 117
******	11	repeated	4, 69, 73, 75, 80, 82, 84, 86, 94, 94, 110
*****	10	keyword	7, 22, 32, 45, 45, 51, 53, 88, 88, 102
*****	10	present	17, 18, 32, 93, 113, 115, 116, 116, 118, 119
*****	9	one	23, 26, 55, 55, 78, 91, 94, 94, 100
*****	9	option is	32, 60, 93, 113, 115, 115, 116, 117, 118
****	8	multi -	25, 27, 32, 48, 48, 50, 108, 118
****	8	number	17, 17, 19, 19, 20, 58, 80, 90
****	8	option is present	32, 93, 113, 115, 115, 116, 117, 118
***	6	keyword - file	7, 22, 32, 45, 51, 102
***	6	listed	50, 114, 115, 116, 117, 118
***	6	multi - tokens	25, 27, 32, 48, 108, 118
***	6	punctuation /	23, 31, 45, 53, 65, 117
***	6	repeated phrase	75, 80, 82, 84, 86, 94
***	5	free format	12, 13, 22, 25, 25
***	5	keywords	23, 31, 46, 65, 117
***	5	list of	22, 24, 25, 53, 84
***	5	output	63, 95, 114, 115, 116
***	5	phrases - file	7, 23, 59, 60, 97
***	5	punctuation - keyword - file	22, 32, 45, 51, 102
***	5	repeated phrases	4, 69, 73, 94, 110
***	5	s	1, 8, 77, 111, 115
***	5	symbolcharacter	26, 40, 43, 45, 50
***	5	tokens - file	8, 25, 32, 48, 108
**	4	case	13, 102, 119, 120
**	4	contains	22, 23, 25, 92
**	4	frequency	69, 71, 77, 78
**	4	ignored - phrases - file	23, 58, 60, 97
**	4	ignored in the tallying	24, 59, 61, 91
**	4	ignored phrase	61, 91, 92, 93
**	4	multi - tokens - file	25, 32, 48, 108
**	4	phrase is	91, 93, 94, 120
**	4	program	30, 57, 98, 111
**	4	punctuation / keywords	23, 31, 65, 117
**	4	sentences	57, 70, 91, 115
**	4	strings	22, 26, 46, 46
**	4	table	69, 70, 73, 86
**	3	b	8, 60, 93
**	3	blank	15, 36, 40
**	3	character	22, 25, 40

```
**              3    file contains                 22, 23, 25
**              3    findphrases                   1, 4, 7
**              3    itself                        82, 86, 92
**              3    line                          14, 23, 24
**              3    maximum                       18, 58, 78
**              3    more                          26, 55, 69
**              3    optional                      17, 23, 25
**              3    phrase which                  92, 92, 93
**              3    phrases in                    4, 57, 110
**              3    sentence                      53, 55, 120
**              3    tables                        75, 95, 114
**              3    tallied                       18, 92, 94
**              3    text                          4, 29, 30
**              3    token                         48, 50, 50
**              3    u                             8, 110, 113
**              3    word                          42, 50, 87
**              3    wordcharacter                 38, 40, 46
*               2    arbitrary text                4, 29
*               2    argument is                   17, 18
*               2    b option is                   60, 93
*               2    blanks                        14, 87
*               2    c                             8, 118
*               2    considered                    58, 88
*               2    consists of                   63, 75
*               2    delimited on each side by     42, 53
*               2    file contains in free format a list of those character strings    22, 25
*               2    files                         12, 100
*               2    given in                      59, 70
*               2    greater than                  20, 94
*               2    ignored phrase is             91, 93
*               2    input                         29, 65
*               2    lines                         65, 84
*               2    listed in the                 50, 114
*               2    m                             32, 117
*               2    maximum length phrase         18, 58
*               2    more than                     26, 69
*               2    multi - token                 48, 50
*               2    newline                       15, 36
*               2    no                            15, 29
*               2    number of                     80, 90
*               2    only the                      94, 113
*               2    particular                    13, 100
*               2    per                           14, 23
*               2    phrase are                    61, 114
*               2    printed                       66, 86
*               2    provided                      31, 59
*               2    punctuation / keyword         45, 53
*               2    range                         19, 105
*               2    repeated phrase itself        82, 86
*               2    repeated phrases in           4, 110
*               2    sequence of                   42, 77
*               2    standard                      29, 63
*               2    strings are called            46, 46
*               2    t                             8, 115
*               2    tab                           15, 36
*               2    table of the repeated phrases 69, 73
*               2    tables of the output          95, 114
*               2    tally                         57, 94
*               2    they                          66, 123
*               2    tokens ( see below            24, 27
*               2    two                           66, 75
*               2    used                          60, 87
*               2    v                             8, 116
```

```
*              2    whatever is in the           45, 48
*              2    whitespace                   36, 42
*              2    zero                         14, 20
```

Output 2. Sorting Example - Version 1: An Informal Strategy.

```
            THE INPUT TEXTFILE

  1  1. Consider the first position in the new list to be created.
  2  2. Consider the first name in the given list which has not yet been crossed out.
  3     Designate it the_earliest_name_in_the_alphabet_which_I_have_so_far_examined.
  4     If this is the only name in the given list go to step 6.
  5  3. Consider the next name in the given list.  If it precedes
  6     the_earliest_name_in_the_alphabet_which_I_have_so_far_examined
  7     in the lexigraphical ordering of names, then let it be the unique
  8     name in the given list to be designated
  9     the_earliest_name_in_the_alphabet_which_I_have_so_far_examined.
 10  4. If the last name considered is not the last name in the given list,
 11     return to step 3.
 12  5. Place the name presently designated
 13     the_earliest_name_in_the_alphabet_which_I_have_so_far_examined
 14     into the position of the new list being considered (the next vacant
 15     position).  Cross out this name in the given list so that it will never
 16     again be considered or compared.  Consider the next position in the new
 17     list.  If at least two names remain in the given list, then return to step 2.
 18  6. Since only one name remains in the given list, transfer it to the
 19     indicated position in the new list, and the alphabetical listing of
 20     the given list will be complete.
 21
     *** End of the Input TextFile ***


            THE REPEATED PHRASES TABLE -- SORTED BY FREQUENCY
Relative
Frequency   Phrase     The Phrase and
Percentage  Count           the Lines Containing It
----------  ------  ---------------------------------
**********    13    list                           1, 2, 4, 5, 8, 10, 14, 15, 17, 17, 18, 19, 20
*******         9    name                           2, 4, 5, 8, 10, 10, 12, 15, 18
*******         9    the given list                 2, 4, 5, 8, 10, 15, 17, 18, 20
******          8    in the given list              2, 4, 5, 8, 10, 15, 17, 18
*****           6    name in the given list         2, 4, 5, 8, 10, 15
****            5    it                             3, 5, 7, 15, 18
****            5    position                       1, 14, 15, 16, 19
***             4    consider the                   1, 2, 5, 16
***             4    the new list                   1, 14, 16, 19
***             4    the earliest name in the alphabet which i have so far examined
                                                    3, 6, 9, 13
**              3    considered                     10, 14, 16
**              3    position in the new list       1, 16, 19
**              3    the next                       5, 14, 16
**              3    to step                        4, 11, 17
**              2    2                              2, 17
**              2    3                              5, 11
**              2    6                              4, 18
**              2    consider the first             1, 2
**              2    consider the next              5, 16
**              2    designated the earliest name in the alphabet which i have so far examined
                                                    8, 12
**              2    names                          7, 17
```

```
**            2    not                          2, 10
**            2    only                         4, 18
**            2    out                          2, 15
**            2    return to step               11, 17
**            2    the last name                10, 10
**            2    this                         4, 15
```

Output 3. Sorting Example - Version 2: With Goto's.

```
           THE INPUT TEXTFILE
  1  Algorithm
  2        Initialize the vacant position to the first position on the new list.
  3    2:  Initialize the given position to the first name not yet crossed out
  4          on the given list.
  5        Get the_name from the given list in the given position.
  6        earliest_alpha_name_so_far <-- the_name
  7        if size of the given list = 1 then
  8          go_to 6
  9        else
 10    3:    Advance the given position.
 11          Get the_name from the given list in the given position.
 12          if the_name preceeds the earliest_alpha_name_so_far then
 13             earliest_alpha_name_so_far <-- the_name
 14          end if
 15          if not at the end_ of the given list then
 16             go_to 3
 17          end if
 18          Insert earliest_alpha_name_so_far in the new list in the vacant position.
 19          Delete the earliest_alpha_name_so_far from the given position.
 20          Advance the vacant position.
 21          if size of the given list > 1 then
 22             go_to 2
 23          end if
 24        end if
 25    6:  Initialize the given position to the first name not yet crossed out
 26          on the given list.
 27        Get the_name from the given list in the given position.
 28        Insert the_name in the new list in the vacant position.
 29    end.
     *** End of the Input TextFile ***


             THE REPEATED PHRASES TABLE -- SORTED BY FREQUENCY
Relative
Frequency   Phrase     The Phrase and
Percentage  Count           the Lines Containing It
----------  ------  ---------------------------------
**********    12    position                                  2, 2, 3, 5, 10, 11, 18, 19, 20, 25,
                                                                27, 28
*********     11    list                                      2, 4, 5, 7, 11, 15, 18, 21, 26, 27,
                                                                28
*******         8   the given list                            4, 5, 7, 11, 15, 21, 26, 27
******          7   the given position                        3, 5, 10, 11, 19, 25, 27
******          7   the name                                  5, 6, 11, 12, 13, 27, 28
****            5   earliest alpha name so far               6, 12, 13, 18, 19
***             4   the vacant position                       2, 18, 20, 28
***             3   get the name from the given list in the given position
                                                              5, 11, 27
***             3   go to                                     8, 16, 22
```

```
***           3    initialize                                  2, 3, 25
***           3    not                                         3, 15, 25
***           3    of the given list                           7, 15, 21
***           3    position to the first                       2, 3, 25
***           3    the new list                                2, 18, 28
**            2    1                                           7, 21
**            2    2                                           3, 22
**            2    3                                           10, 16
**            2    6                                           8, 25
**            2    advance the                                 10, 20
**            2    earliest alpha name so far <-- the name     6, 13
**            2    in the new list in the vacant position      18, 28
**            2    initialize the given position to the first name not yet crossed out on the
                        given list
                                                               3, 25
**            2    insert                                      18, 28
**            2    size of the given list                      7, 21
**            2    the earliest alpha name so far              12, 19
```

Output 4: Sorting Example - Version 3: With Goto's Removed.

```
          THE INPUT TEXTFILE

 1  Algorithm
 2      the_vacant_pos := first_position_on_list (the_new_list);
 3      while size_of (the_given_list) > 1 do
 4         the_given_pos
 5                := position_of_first_name_not_crossed_out (the_given_list);
 6         the_name := get_name (the_given_list, the_given_pos);
 7         earliest_alpha_name_so_far := the_name;
 8         while not end_of_list (the_given_list) do
 9            advance (the_given_pos);
10            the_name := get_name (the_given_list, the_given_pos);
11            if name_precedes (the_name, earliest_alpha_name_so_far) then
12               earliest_alpha_name_so_far := the_name
13            end if
14         end while
15         insert (earliest_alpha_name_so_far, the_new_list, the_vacant_pos);
16         delete (earliest_alpha_name_so_far, the_given_list);
17         advance (the_vacant_pos);
18      end while
19      the_given_pos := position_of_first_name_not_crossed_out (the_given_list);
20      the_name := get_name (the_given_list, the_given_pos);
21      insert (the_name, the_new_list, the_vacant_pos);
22  end.
23
    *** End of the Input TextFile ***


           THE REPEATED PHRASES TABLE -- SORTED BY FREQUENCY

Relative
Frequency   Phrase     The Phrase and
Percentage  Count           the Lines Containing It
----------  ------  ---------------------------------
**********     8    :=                                  2, 5, 6, 7, 10, 12, 19, 20
**********     8    the given list                      3, 5, 6, 8, 10, 16, 19, 20
*********      7    ( the given list                    3, 5, 6, 8, 10, 19, 20
*********      7    the name                            6, 7, 10, 11, 12, 20, 21
********       6    the given pos                       4, 6, 9, 10, 19, 20
```

```
******         5    earliest alpha name so far               7, 11, 12, 15, 16
******         5    the given list )                         3, 5, 8, 16, 19
*****          4    ( the given list )                       3, 5, 8, 19
*****          4    the given pos )                          6, 9, 10, 20
*****          4    the vacant pos                           2, 15, 17, 21
****           3    the name := get name ( the given list , the given pos )
                                                             6, 10, 20
****           3    the new list                             2, 15, 21
****           3    the vacant pos )                         15, 17, 21
***            2    ( earliest alpha name so far ,           15, 16
***            2    ( the name ,                             11, 21
***            2    , the new list , the vacant pos )        15, 21
***            2    advance (                                9, 17
***            2    earliest alpha name so far := the name   7, 12
***            2    insert (                                 15, 21
***            2    the given pos := position of first name not crossed out ( the given list )
                                                             4, 19
```

Output 5. Sorting Example - Version 4: Part of a Pascal Implementation.

```
           THE INPUT TEXTFILE

  1  (* Sort the Given List of Names *)
  2     TheVacantPos := FirstPositionOnList (TheNewList);
  3     while (SizeOfList (TheGivenList) > 1) do begin
  4        TheGivenPos := PositionOfFirstNameNotCrossedOut (TheGivenList);
  5        GetName (TheName, TheGivenList, TheGivenPos);
  6        EarliestAlphaNameSoFar := TheName;
  7        while (not EndOfList (TheGivenPos)) do begin
  8           Advance (TheGivenPos);
  9           GetName (TheName, TheGivenList, TheGivenPos);
 10           if (NamePrecedes (TheName, EarliestAlphaNameSoFar)) then
 11              EarliestAlphaNameSoFar := TheName;
 12           { end if }
 13        end; { while }
 14        Insert (EarliestAlphaNameSoFar, TheNewList, TheVacantPos);
 15        Delete (EarliestAlphaNameSoFar, TheGivenList);
 16        Advance (TheVacantPos);
 17     end; { while }
 18     TheGivenPos := PositionOfFirstNameNotCrossedOut (TheGivenList);
 19     GetName (TheName, TheGivenList, TheGivenPos);
 20     Insert (TheName, TheNewList, TheVacantPos);
 21     Delete (TheName, TheGivenList);
 22
     *** End of the Input TextFile ***


            THE REPEATED PHRASES TABLE -- SORTED BY FREQUENCY
Relative
Frequency   Phrase     The Phrase and
Percentage  Count           the Lines Containing It
----------  ------  ---------------------------------
**********     8    TheGivenList                     3, 4, 5, 9, 15, 18, 19, 21
**********     8    TheName                          5, 6, 9, 10, 11, 19, 20, 21
*********      7    TheGivenPos                      4, 5, 7, 8, 9, 18, 19
********       6    TheName ,                        5, 9, 10, 19, 20, 21
******         5    :=                               2, 4, 6, 11, 18
******         5    EarliestAlphaNameSoFar           6, 10, 11, 14, 15
******         5    TheGivenList )                   3, 4, 15, 18, 21
******         5    TheGivenPos )                    5, 7, 8, 9, 19
```

```
*****          4    TheName , TheGivenList             5, 9, 19, 21
*****          4    TheVacantPos                       2, 14, 16, 20
****           3    GetName ( TheName , TheGivenList , TheGivenPos )
                                                       5, 9, 19
****           3    TheNewList                         2, 14, 20
****           3    TheVacantPos )                     14, 16, 20
***            2    Advance (                          8, 16
***            2    Delete (                           15, 21
***            2    EarliestAlphaNameSoFar ,           14, 15
***            2    EarliestAlphaNameSoFar := TheName  6, 11
***            2    Insert (                           14, 20
***            2    TheGivenPos := PositionOfFirstNameNotCrossedOut ( TheGivenList )
                                                       4, 18
***            2    TheNewList , TheVacantPos )        14, 20
```