Educational Materials
CMU/SEI-90-EM-3
August 1990

Reading Computer Programs: Instructor's Guide and Exercises

Lionel E. Deimel

Software Engineering Curriculum Project

J. Fernando Naveda

University of Scranton

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This document was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this document should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This document has been reviewed and is approved for publication.

FOR THE COMMANDER

JOHN S. HERMAN, Capt, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Table of Contents

Reading Computer Programs: Instructor's Guide and Exercises

Abstract: The ability to read and understand a computer program is a critical skill for the software developer, yet this skill is seldom developed in any systematic way in the education or training of software professionals. These materials discuss the importance of program reading, and review what is known about reading strategies and other factors affecting comprehension. These materials also include reading exercises for a modest Ada program and discuss how educators can structure additional exercises to enhance program reading skills.

1. Introduction

This report has two main objectives: to convince teachers of future computer professionals of the importance of program reading, and to provide sample exercises to facilitate the teaching of program reading. We will review the literature relevant to program reading, and discuss teaching approaches and techniques. A large part of the report is devoted to the listing of an Ada program for which we provide reading exercises that fulfill a range of educational objectives. The program and exercise questions are available on diskette from the Software Engineering Institute (SEI). An order form for machine-readable versions of this material is provided at the end of the report. Readers are encouraged to adapt the materials to their own particular needs.

Although the reading exercises are based on the Ada programming language, it would be a mistake to view this as an "Ada report." Most of what we will say about program reading is independent of any particular language, although we make no special attempt to discuss languages other than high-level procedural ones. Ada *is* an attractive vehicle to illustrate reading exercises because it contains good mechanisms for concurrency, information hiding, and the like. Exercises are provided to explore these facilities and to explore the Ada language generally. The reader whose students are not using Ada can nonetheless benefit from the ideas we present, including our suggestions for generating reading exercises from materials other than those provided here.

1.1. What Is Program Reading?

By *program reading* (or *code reading*), we mean the process of taking computer source code and, in some way, coming to "understand" it. We mean to make no presuppositions about whether or not the code contains internal documentation (comments) or whether any form of external documentation (design, structure charts, etc.) is available to the reader or can be easily created.

Presumably, some of what can be said about code reading also applies to attempts to understand other documents—software specifications or designs, for example. We will

not, however, attempt to generalize to these other life-cycle products, in part because of their diversity of representation. Except where we note otherwise, we mean for our remarks to be applied only to the realm of high-level procedural languages, although much of what we will say applies to code outside this class.

We will have more to say about what we mean by "program understanding." For now, it is important to realize that we do not often read programs for what might be intuitively described as "total understanding." We usually read programs for particular purposes—to determine if they have particular properties such as correctness, or to discover how an enhancement can be made with minimal disruption to the program's current functionality. When we speak of reading a program, we refer to a process that aims to achieve whatever degree of understanding is needed to accomplish our particular objectives.

1.2. Overview of This Report

In the next chapter, we argue the importance of developing program reading skills among software developers. We also mention educational issues associated with program reading, issues we will deal with at greater length in Chapter 5.

In Chapter 3, "How Do People Read Programs?," we discuss theoretical and empirical research on program reading, and examine proposed models of program comprehension. We try to answer questions such as: "What does it mean to understand a program?" and "How do programmers ascertain the meaning of a program?" The chapter provides useful background for the instructor who wants to teach program reading skills. Some of this material should also be shared with students in lectures.

Chapter 4, "Readability Factors and Tool Support," discusses factors that affect the ease with which a program can be understood. Much stylistic advice to programmers in the literature is implicitly based on assumptions about what makes a program readable. Likewise, observations about factors that make a program readable imply corresponding style rules. In this chapter we look at the relation of empirical observations to programming standards. We also discuss tools that can make program reading easier, now or in the future.

In "Teaching Program Reading," Chapter 5, we assert that program reading is a skill that should be taught explicitly. We discuss what can be taught and how. We also look at other uses of program reading in the classroom, whereby students can develop their reading skills in the pursuit of other educational objectives.

Chapter 6, "Constructing Reading Exercises, with Examples," contains a discussion of how to construct program reading exercises. The chapter also contains specific exercises targeted to a variety of educational objectives and based on the concurrent Ada program listed in the Appendix.

We have included an annotated bibliography, which contains not only references cited in the text, but also related literature. We have tried to indicate the significance of each entry and suggest how it might be used in teaching. In most cases, we have sum-

marized the content of the entry. These annotations often elaborate on ideas treated only briefly in the text. The bibliography is by no means complete, but we believe that interested readers can locate other reading-related papers and monographs that might be of interest to them by reading the references we have included, many of which themselves have excellent bibliographies.

2. The Importance of Program Reading Skills

The successful programmer must master a number of diverse skills. One that is often overlooked is the ability to read and understand a program. Reading competence is most obviously relevant to program maintenance, for which the programmer must gain sufficient understanding of the code to design modifications that extend, adapt, or correct it, while retaining its logical, functional, and stylistic integrity. Since, typically, more than half the resources devoted to a program over its lifetime is expended on maintenance, and since reading the program can be an important step in the maintenance process, lack of adequate reading skills among maintenance personnel can have serious financial consequences.

But program reading ability is also important in non-maintenance activities. Effective code verification and code reviews require reviewers to understand and analyze the code under scrutiny. The expert reader is a valuable team member in such circumstances, as well as a resource to colleagues when they are having a difficult time debugging.

Code reading provides an all-too-infrequent opportunity for programmers to share and learn from one another's work. Code examined in formal reviews and in program libraries and repositories; published programs; and code found in professional journals all offer the programmer the chance to deepen his understanding of his craft and improve his skills.[1] Learning a new programming language is often facilitated by reading existing code written in that language by more experienced programmers. Studying programs in this way allows the programmer to gain a sense of proper idiom and style more quickly and surely than does studying language reference manuals, style guides, and coding standards.

A programmer who reads programs effectively can increase his code generation efficiency. He is better able to find and evaluate code that can be reused or adapted than are his less program-literate colleagues. This ability allows him to avoid a good deal of coding altogether. When working on a large software system, a programmer needs to be reminded of the details of code written so long ago that those details are no longer fresh in his mind. Facility in reconstructing such information from code, even one's own, can therefore be a valuable asset.

Students in academic programs do get occasional opportunities to practice reading programs. They are presented with code in introductory and data structures textbooks, and possibly in textbooks used in courses on other topics, such as analysis and design of algorithms. Students are sometimes given example programs by their instructors and may have to struggle to make sense of program units written by fellow students on team projects, such as might be part of an introductory software engineering course. Rarely, examinations require students to interpret code.

---

[1] We will follow the traditional convention of using masculine terms where the person spoken of may be either male or female. No offense to female professionals—who are distressingly underrepresented in the workplace—is intended, of course.

Nonetheless, it is fair to say that the improvement of program reading skills is virtually unknown as an explicit educational objective among those teaching future software developers. Yet the available evidence suggests that our current neglect of the topic cannot be justified by the argument that adequate program reading skills develop naturally and without special encouragement in students otherwise well prepared to enter professional practice.

We advocate that instructors undertake activities designed to teach and improve program reading skills among their students. Although this suggestion is radical and largely untried, substantial benefits might be gained. Not only is there potential for graduating students better able to perform important tasks such as program maintenance, but there is also a realistic hope that students will become better learners in advanced courses by virtue of their greater ability to understand program examples that illustrate specialized techniques. Such students should also be better learners on the job. Classes of competent program readers can be given examination questions to test their program-writing skills more reliably by means of questions based on program fragments that must be read.

No doubt, many educators will object to our suggestion of teaching program reading, protesting either that educators have no idea how to carry out such a suggestion or that they have no time to do so. We will try to counter these arguments by enumerating techniques that can be used in the classroom, including activities that use program reading incidental to achieving unrelated educational objectives. We must point out that, although no one disputes the importance of teaching program writing skills, there is hardly a consensus about the most effective way of doing it. Reading skills seem equally important, and our lack of educational experience teaching them seems a poor excuse for continuing to ignore program reading in the classroom.

2.1. Growing Awareness of Program Reading Issues

Program reading seems to be attracting increasing attention, largely because of its pivotal role in maintenance. As early as 1971, however, Gerald Weinberg lamented the decline in the practice of reading programs brought about by time-sharing [Weinberg71]. He was less concerned with the role of reading in maintenance than with its potential for teaching; programmers could learn a good deal from reading the programs of others, as well as their own. In the preface to their well-known book, *The Elements of Programming Style*, Kernighan and Plauger make a similar point about the educational value of reading programs [Kernighan74]. Knuth was so taken with the benefits of program reading that he designed a programming scheme, literate programming, that recognizes the human reader to be as important as the mechanical one [Knuth84]. He has published two programs in this form, each of which is a sizable book [Knuth86a, Knuth86b]. David Moffat published a more modest collection of program readings in Pascal for the beginning student [Moffat84] and suggested, with Lionel Deimel, that program reading can play a central role in the teaching of programming generally [Deimel82].

Much of the literature on program reading deals with *program comprehension*, which encompasses both what it means to comprehend a program and the process by which that comprehension is reached. Lukey [Lukey81] lists three approaches to studying program comprehension—the *observational approach*, the *experimental approach*, and the *artificial intelligence approach*. Roughly, these approaches correspond to the study of programmers reading programs to see what they do, the study of programmers reading in situations where certain factors are fixed and others are allowed to vary in controlled ways, and the production of tools that serve to mechanize or assist in the program comprehension process. Papers describing each approach have not only increased in number over the years, but also are now more likely to appear in such "mainstream" computer journals as *Communications of the ACM*.

Of course, observations of what makes a program readable have implications for how programs should be written and formatted. In addition to a number of books advocating a certain programming style (*e.g.*, [Kernighan81]), some authors are making more radical proposals affecting the visual display of programs [Baecker90, Oman90a, Oman90b], or are looking to environments that support program reading in a more organic way [Goldberg87].

We consider it important and exciting that both the Florida/Purdue Software Engineering Research Center (SERC) [Wilde89] and IBM [Corbi89] have undertaken significant efforts to study program comprehension and to explore tools to assist programmers in reading programs effectively. We expect such tools eventually to become commonplace, although they will not obviate the need for programmers to have good program reading skills of their own in order to use these tools effectively.

3. How Do People Read Programs?

What do we know about how people read or should read programs? Several writers have offered advice about how we should go about reading code. Through experimental and observational studies, we have indications of how actual programmers behave when trying to understand unfamiliar code. Finally, there have been many attempts to construct models of program understanding, both to explain empirical findings and to make useful and verifiable inferences about how program reading can be carried out more effectively. Let us look at models of program understanding or, as they are usually called, "program comprehension" models.

3.1. What Is Program Understanding?

If we are to construct a useful theory of program reading, we require a model to help us explain how we come to understand a program, as well as what we can be said to know when we do understand that program.[2] There must be a computer science component in any such model, of course, but there are also behavioral and psychological components. Many researchers in the field of program reading have approached their studies from a cognitive psychology or artificial intelligence point of view. Although this is not remarkable, we point it out to alert the reader that the study of program reading, while important to software engineering, is not strictly an area of computer science research.

Let us first look at what we know when we understand a program. Almost any programmer could generate a useful list. We understand what each statement means, how flow of control passes from one statement to another, what algorithms have been employed, how information is represented and transformed in data structures, which subprograms invoke other subprograms, and how the program interacts with its environment. Ruven Brooks has described all this information as a succession of "knowledge domains" that bridge between the problem domain and the executing program [Brooks78, Brooks82, Brooks83]. A knowledge domain is a collection of information about objects of some sort and relationships among those objects. (One may think of this in the mathematical sense of a set of objects, a collection of relations, and so forth.) According to Brooks, the knowledge domains include (or may include): the problem domain; the domain of some mathematical model for the real-world problem; an algorithmic domain of abstract data structures and operations; an implementation domain of arrays, assignments, and the like; and a domain of bit patterns stored and manipulated in specific storage locations within the computer. (It is easy to argue about the boundaries of these domains and whether there are more or fewer of interest. Certainly the program structure domain—which includes objects like procedures, functions, and tasks—deserves mention.) If one really understands a program, according to this theory, one possesses the knowledge of each of the domains, as well as the ability, through knowledge of inter-domain relations, to relate conceptionally adjacent domains

[2] We use the term "program" to mean any appropriate piece of code, be it a complete program, a subprogram, or even an isolated segment.

to one another. The process of understanding a program is one of constructing (or reconstructing) the knowledge domains and relations among them from the code, comments, and whatever other documentation is available.

Even in the absence of theoretical or empirical underpinnings, Brooks's model is intuitively appealing. It seems to capture succinctly most of what might reasonably be construed as knowledge about a program, and it suggests particular ways in which our knowledge can be incomplete. For example, we may understand a program perfectly statement-by-statement, yet fail totally to comprehend, on a more abstract level, what it does and why. Brooks has used his model to make inferences about documentation. Effective documentation communicates information not explicitly present in the source code itself (*i.e.*, it adds to the information available to the reader). Therefore, languages like traditional FORTRAN may require more explanation of their code than languages like Pascal, which allow direct manipulation of higher-level abstractions [Brooks82].

Another point made by Brooks deserves special mention, namely that the code itself (*i.e.*, the actual source-language statements) is *not* the ultimate authority with respect to program meaning. On one hand, this statement is surprising. A program designates some computation, which may be taken to be its semantics or meaning. Therefore, given a well-defined programming language and computing platform, a syntactically correct program definitively specifies what the program does. It is this point of view, as expressed by Kernighan and Plauger in *The Elements of Programming Style* [Kernighan74], that Brooks attempts to counter. Kernighan and Plauger argue that a program and its documentation provide multiple representations that are subject to inconsistencies and that, therefore, "the only reliable documentation of a computer program is the code itself." Brooks argues, in effect, that such a position ignores the larger context in which programs exist. To begin with, a program's function is partly a matter of the interpretation given the input and output, information perhaps only hinted at in the code. Further, for purposes such as maintenance, explicit statements about assumptions and design decisions that led to the source code may be more important than the code itself for the programmer trying to understand the program.

Several authors have proposed models similar to that of Ruven Brooks. The differences among these models seem mostly confined to minor details and nomenclature. In any case, the layered structure of one's knowledge of a program is a common element in all these theories.

Shneiderman and Mayer have proposed a "syntactic/semantic" model of programmer behavior [Shneiderman79]. The model is intended to apply to writing programs, as well as reading them, modifying them, and learning to program. The model assumes that semantic and syntactic knowledge is stored in long-term memory and manipulated in short-term memory and working memory. Program comprehension, they suggest, is largely a matter of building up a hierarchy of semantic knowledge about the program, with information about what the program does at the top of the hierarchy, and lower-level information about statements and algorithms below. The representation is in terms of abstractions (representing the function of groups of statements, for example) derived from the program text. All the mental machinery of the syntactic/semantic

model seems less useful than one might hope, as we simply do not know enough details of how programmers perform. The state of understanding a program, however, seems quite similar to that postulated by Brooks.

Letovsky refines some of Brooks's notions and speaks of the reader's *knowledge base* (presumably the reader's syntactic and semantic knowledge) and *mental model* of the program being read [Letovsky86a]. In referring to the reader as a *knowledge-based understander*, he emphasizes what is implicit in Brooks's discussions but is more prominent in the model of Shneiderman and Mayer, namely that the reader brings a good deal of expert knowledge to his task. Letovsky brings something of an artificial intelligence perspective to what knowledge the reader has and what his mental model looks like. According to him, when a reader has a complete understanding of a program, he possesses a description of the goals of the program, the actions and data structures of the implementation, and an explanation of how goals or subgoals are accomplished by the components of the implementation. His hierarchy is one of goals and subgoals. All of these models, however, involve layers of knowledge that become progressively more abstract and that are ultimately tied to larger and larger fragments of the program.

3.2. Program Reading Strategies

The question "How do you read a program?" may seem strange to a beginning student. Yet, the obvious answer, "from top to bottom," is hardly the correct one. To begin with, the sequence of statements in the text of a program does not correspond in a straightforward way to the order in which the statements are executed when the program is running. This correspondence is complicated by the syntactic rules that determine the placement of subprograms and other definitions, and by statements that alter the sequential flow of control. Additionally, the execution path through a program is, in general, a function of the input data; it therefore differs from one invocation of the program to the next.

These statements may seem unremarkable. Because we believe students should be taught early how to read programs, however, it is important to make the statements, lest students develop fundamental misconceptions at the outset. Programs are not read like novels, nor is their meaning determined by seeing how they behave when run or hand-traced using test data. Building from a program the layered abstractions that become one's mental model of it—Letovsky calls the process *assimilation* [Letovsky86a]—is a complex task that is only beginning to be understood. How *does* the program reader construct a mental model of a program, whatever the exact nature of that model might be?

To begin with, there are many factors that may simplify assimilation of a program or make it more difficult: the knowledge and experience of the reader, the complexity of the algorithms used, the programming language, the use of structured programming techniques, the presence and quality of comments in the code, the availability of external documentation, etc. We will discuss these factors briefly in Chapter 4. Insofar as possible, we will gloss over them in the discussion that follows.

Empirical evidence suggests that the human program reader is what Letovsky calls an "opportunistic processor," capable of using multiple reading strategies [Letovsky86a]. The basic strategies people have written about, however, are *top-down* and *bottom-up*.

Top-down reading of a program is analogous to the more familiar top-down development approach. One begins by gaining an understanding of the overall purpose of the program, then tries to understand how that function is implemented by component pieces. In carrying through this process, the reader forms hypotheses about these pieces—they may be procedure calls, individual loops, or other fragments—which are later verified or modified through recursive application of this method. Letovsky describes this approach as building a representation of the specification first and then working down to the implementation level.

Reading bottom-up proceeds in the opposite direction. Understanding of small fragments of code is aggregated into descriptions of larger pieces of the program until the overall program function and strategy have been discovered.

It is worth noting here that two characteristics of a program have a dominant influence on reading approaches that may be available: the degree of documentation and whether the program is structured. Linger, Mills, and Witt point out that poorly documented code generally must be read bottom-up, as the lack of documentation can make it nearly impossible to devise hypotheses about what various sections of code accomplish without examining those sections of code in detail. Well-documented code can usually be read top-down [Linger79].

In explaining how the function of a structured program can be ascertained from an understanding of its primitive components, Linger, Mills, and Witt also explain why unstructured programs are difficult to understand. Unstructured code cannot easily be resolved into components that interact with one another in a limited number of simple ways. Although the program comprehension literature discusses programs having various levels of documentation, we are aware of virtually no work done on unstructured programs. Researchers advise readers to restructure their programs algorithmically and then read the newly structured program [Basili82, Linger79].[3] Tools are available to perform restructuring, at least for some languages, such as COBOL.

The bottom-up approach to understanding a program is perhaps the easier of the two basic strategies to explain. It is the simpler and, in the sense that it can be applied with success in the most situations, the most general. Basili and Mills show the application of a bottom-up approach to program understanding in [Basili82]. Their approach is very formal, but the formality is inessential to understanding the basic strategy. After converting their program into a structured program, they begin to assign meaning (function) to each *prime program*, each structured unit such as an **if** ... **then** ... **else** (see [Linger79]). This assignment of meaning may be done through "direct cognition," or

---

[3]Because much unstructured code still exists, we can ask if we should teach our students how to deal with it. Probably advocating the procedure described here is the best approach. Surely unstructured code can be read, and even maintained, but procedures for reading and maintenance are necessarily *ad hoc*.

it may require more deliberate analysis. Since a structured program is composed only of sequenced and nested structured units, the function of larger fragments of the program may be discovered by combining the functional descriptions of smaller program components in standard ways. Carrying this process through, one arrives at a functional description for the entire program, as well as additional information collected along the way. Linger, Mills, and Witt call this strategy *stepwise abstraction*, and it is certainly a process programmers carry out all the time, particularly for small segments of code.

Shneiderman describes program comprehension as proceeding in a similar way, though he substitutes the language of psychology for that of mathematics. He describes the reader as recognizing the function of groups of statements as "chunks" and combining these chunks to explain larger program fragments [Shneiderman79].

In Brooks's model, the primary assimilation process proceeds top-down. Brooks obliquely acknowledges that people do use bottom-up strategies, but he dismisses them as less powerful [Brooks83]. The process Brooks describes is one of repeated hypothesis generation and verification. Knowing whatever he knows about the code, the reader generates hypotheses about what the program does and how. He tries to verify these hypotheses by examining the code. Evidence that feeds hypothesis generation comes from the program text, program comments, and whatever additional documentation may be available. If the reader sees a variable named `DIST_TBL`, for example, he might conclude that it probably stores a table of distances between locations. He would then try to verify this hypothesis by further examining the code. Hypothesis generation begins with what is known about the overall function of the program, which often by itself causes the reader to expect to see certain features in the program—sorts, master and transaction files, and the like. Hypotheses tend to be non-specific and therefore usually are not directly verifiable. Instead, they cause subsidiary hypotheses to be generated in a hierarchical fashion until a level is reached at which hypotheses can be directly verified or proven false. The reader scans the code in various ways searching for clues in the text that bear on current hypotheses. In this scanning process, he looks for "beacons," which are typical evidence for certain structures or operations. Two nested loops might be a beacon for a sort, for example. The program is understood when verified hypotheses have been bound to all code in the program.

Several empirical studies have added to our understanding of what programmers actually do when reading programs. Videotaped protocols of professional programmers modifying a program are the basis of papers by Letovsky [Letovsky86a] and Littman, Pinto, Letovsky, and Soloway [Littman86]. Both these papers contain extensive descriptions and analyses of what the subjects did. The data show quite clearly that they do not employ pure top-down or bottom-up strategies, but freely mix the two. Insight into why this should be so comes from Letovsky's paper, in which he classifies questions the subjects posed to themselves. For example, a "how" question ("So let's see how it searches the database.") is top-down in nature; a lower-level implementation is sought for a goal. A "why" question ("It's setting `IPTR` to zero. I'd like to know why.") is looking in the other direction; a goal is sought for a part of the implementation. Other questions discussed in the paper are "what" questions ("I want to find out what field 7 is.") and "whether" questions ("Is this subroutine actually deleting a record or is it just

putting a delete mark there, and the record is still there?"). Programmers are clearly seen here engaged in the sort of activity Brooks describes. When seen in its detail, however, the process is not simply characterized. Letovsky also describes particular techniques programmers use to generate hypotheses (he calls them conjectures) to answer their questions.

Littman, *et al.*, emphasize a different aspect of reading strategy. They report that subjects in their study used either what they call a "systematic" strategy or an "as-needed" strategy. Subjects either tried to completely understand the program being modified, or else they tried to gain just enough knowledge to make the change requested in the program. This raises the question of reading objectives; not always—perhaps not even often—does the reader really need to know everything about a program. One's approach to reading is presumably conditioned by one's purpose for reading. This paper provides a warning, however. Subjects who used the systematic strategy, employing extensive symbolic execution of the code, successfully modified the program. Those who cut corners with an as-needed strategy were unsuccessful at the modification task, as they failed to detect critical interactions among program components. This is a distressing outcome, as truly large programs may not be amenable to total understanding.

Although the Letovsky and Littman papers suggest that actual comprehension strategies are eclectic,[4] they do not cast doubt on the layered mental models discussed in the last section. A similar message can be inferred from [Pennington87]. Pennington found that readers who focus on both the problem domain and the program domain, as opposed to focusing on one or the other, are more successful, a result one might reasonably expect from, say Brooks's model of comprehension.

Several of the references contain brief but useful reviews of at least part of the program comprehension literature. See, for example, [Baecker90], [Corbi89], [Crosby90], and [Wilde90].

[4]Students should understand both top-down and bottom-up strategies, and should be familiar with descriptions of the eclectic processing that is apparently typical. Knowing about alternative strategies should help students select appropriate ones in particular situations.

4. Readability Factors and Tool Support

In this chapter, we further examine the pragmatics of reading programs. What are the factors—either internal or external to the program—that affect how difficult a program is to read? How does the style with which a program is written affect its readability, and how can observations about readability be translated into useful coding style guidelines? We will also examine reading strategies in more detail and see how existing and future software tools can aid the program reader.

There is inadequate space here to treat these topics thoroughly. There are, however, few actual facts to cite. Our knowledge is tantalizing and fragmentary, with most of the obviously interesting questions having ambiguous answers (how long should a procedure be?) or none at all (what is the most readable programming language?). Yet software development is a practical enterprise, not a theoretical one; we cannot wait for the answers. We must look for credible insights wherever they are to be found, and forge ahead.[5]

4.1. Factors Affecting Program Readability

The study of program comprehension naturally leads us to the question of what makes a program readable. We do not have a complete theory that allows us to predict the readability of a program, but it is not difficult to offer a list of parameters that might be important. Some of these factors have been investigated systematically, some not. (The recently published bibliography by Thomas and Oman contains references to many of the factors discussed below [Thomas90].) We may classify these parameters into the following categories:

1. Reader characteristics.
2. Intrinsic factors.
3. Representational factors.
4. Typographic factors.
5. Environmental factors.

It should be clear from the previous chapter that the reader has a great effect on how successfully a particular program is read. Comprehension models suggest that the reader's knowledge—of programming, of the programming language, and of the application domain—as well as his reading strategy are important variables [Brooks83]. The reading strategy is sometimes a function of the programmer's purpose in reading. For example, the as-needed strategy in response to limited goals, described by Littman, *et al.*, in [Littman86], led directly to comprehension errors. In [Letovsky86b], Letovsky and Soloway describe how the unwillingness of readers to search for information not imme-

[5]We did this in the case of structured programming, for instance. Although structured programs have certain logical advantages over non-structured ones, we have never *proved* that structured programming is superior. Even the hypothesis is ill-defined.

diately near the code they are examining predictably leads to comprehension errors. Perhaps because program reading skills have not traditionally been taught, more than one study have included statements like the following from [Littman86]:

> Finally, we note that there was virtually no relationship between years of professional programming experience and either successfully performing the enhancement task or the programmer's choice of study strategy.

Perhaps this is another manifestation of the well-known disparity in efficiency among professional programmers.

It is reasonable to suppose that programs possess a greater or lesser degree of intrinsic complexity, which affects their readability. It is difficult to say how to measure this complexity, and, for this reason, the field of software metrics is controversial. Surely a "hello, world" program is less intrinsically complex than the average program to compute Bessel functions. Moreover, concurrent or real-time programs are probably more complex than comparable programs without these characteristics, and very large programs are difficult to read simply by virtue of their size. Brooks notes that there are abstruse problems that have simple programs as their solutions; the nature of the problem may still make these programs difficult to comprehend [Brooks83]. Shneiderman discusses logical, structural, and psychological complexity of programs and their relation to comprehensibility in [Shneiderman80].

The term "representational factors" is deliberately broad. It is easy to subdivide, though it can be difficult to make clear distinctions among the resulting categories. We mean to include such parameters as the programming language, the nature and inclusion of comments, the architectural structure of the program, the choice of identifiers, and other choices that need to be made to generate the *logical* program, while excluding those choices involved with the relative placement of program characters, which we put under "typographic factors." Many of these factors are discussed in [Shneiderman80]. Brooks [Brooks83] makes some interesting remarks about documentation, which apply to internal comments. He suggests that:

- Different kinds of documentation are helpful at different stages of comprehension. In the early stages, high-level program descriptions are helpful, whereas in later stages, lower-level information is more useful.
- Different languages need different kinds of documentation.
- Too much documentation can obscure as much as it can illuminate.

Typographic factors include the use of upper- and lowercase, fonts, color, and white space. We mean to distinguish between the logical program and the representation of it presented to the user. This allows us to include in this category rearrangements of the code such as that effected by literate programming systems [Knuth84]. Interesting studies have been made concerning typographic factors. One of these was done by Miara, Musselman, Navarro, and Shneiderman [Miara83]. In this carefully done study, the authors concluded that indentation to show structure does enhance program readability, though it is possible to have either too little or too much indentation. Other studies can be found in the bibliography of this document, in [Shneiderman80], or in the Thomas and Oman bibliography [Thomas90].

Our term “environmental factors” is also somewhat of a catch-all. We mean to include both the physical and the logical environment in which reading takes place. One may place room temperature in this category for completeness, but we have in mind particularly factors, such as: the medium of the program (paper, CRT monitor); external documentation; and software tools, like editors and compilers. Interestingly, in this age of high-definition workstation monitors, there are many circumstances in which programmers still prefer using paper listings, and presumably are more efficient workers for doing so [Oman90a].

4.2. Readability and Style

A good deal has been written, much of it atheoretical, about programming style, a term denoting an array of software-writing practices involving choice of identifiers, effective use of language features, commenting conventions, indentation, use of white space, use of case and font, and the like. To the degree that we learn about what makes a program readable, we should be able to turn this knowledge around and make it into programming style guidelines that should produce more readable programs. Instructors who want to teach program reading should be sure their students make this connection. The reader is directed to [Baecker90], [Oman90b], [Shneiderman80], and [Thomas90].

4.3. Tools and Techniques

As we have seen, actual program reading behavior is complex. Although we cannot offer a general plan for reading programs, we can provide hints, suggest simple tools that can be used, and catch a glimpse of automated support that might be available in the future.

We begin with a list of ideas for program reading, designed more for suggesting what sort of advice we can offer our students than as a complete bag of tricks. Readers can no doubt add their own entries to our list:

1. Be aware that code and comments (or other documentation) may not agree. The code may be correct and the comments wrong, or the reverse. Both may be wrong. (The code may not accomplish what it is supposed to do, and the comments may describe neither what should be done nor what is done.)

2. Use indentation to help understand structure. However, incorrect indentation (more likely to come from a human than a compiler or prettyprinter) may be misleading.

3. Try to build a model of the style conventions used in the program. If, for example, a consistent scheme has been used for identifiers, this knowledge can be used to help understand the meaning of newly encountered identifiers. It is important to read the program with the programmer’s conventions, rather than your own, in mind.

4. Be wary of apparently analogous functions performed in non-analogous ways. Arbitrary or stylistic differences may simply indicate programmer inconsistency, but they may also signal modified code (a maintainer with different habits has modified the code) or code whose functions are not as analogous as they might at first appear.

5. Consider the possibility that the programmer did not know what he was doing.

6. Watch out for code written to overcome compiler or computer limitations or code containing apparently *magic* numbers.

7. Watch out for use of nonstandard language features. (Some compilers, for example, initialize variables that other compilers do not.)

8. Use stepwise abstraction.

9. Odd-looking arithmetic operations may be required to maintain accuracy. Consider possible roundoff implications.

10. Because changes to the code often introduce errors and inconsistencies, look for evidence of changes. Look for change logs or comments about changes imbedded in comments. Stylistic differences can indicate changes by a programmer other than the author. If multiple versions of a program exist, using a tool to find changes (*e.g.*, UNIX `diff`) can be helpful.

11. If, after a good deal of study, a piece of code is making no sense, ask another programmer to look at it. Consider explaining to him what you think you *do* know.

12. Search for information, particularly in documentation, that relates objects in different knowledge domains, for example, comments that associate variables with problem-domain objects.

13. Be wary of objects that have the same identifier but different scopes. Reasoning about the wrong objects can be frustrating.

14. Be wary of objects having *nearly* the same names, particularly those whose identifiers differ by a single character.

15. Particular code may be an artifact that no longer serves a function.

16. Be sure you make no inessential assumptions when reasoning about concurrent programs.

17. Be alert for variables that serve more than one function or that are used inconsistently, as they can mislead the reader.

18. The effect of apparent bugs in the program can be undone by an inverse bug somewhere else.

19. Use symbolic execution to determine function.

20. Use code substitution to verify or refine hypotheses. Substitute code for what you *think* is being performed into the program, and examine how your code differs from what code is actually there.

21. Tracing code with test data, whether by hand or using a symbolic debugger, will not by itself tell you what function the code performs. However, it can help suggest some hypotheses and eliminate others.

22. Be alert for literals that are conceptually distinct but that happen to have the same values. (The trouble usually begins when one tries to modify such code.)

23. In languages that permit operator overloading, be sure the operator you think you have is really the one you do have.

24. Be willing to abandon hypotheses for which there is insufficient evidence.

25. Use *program slicing* [Weiser81]. Throw out parts of the program irrelevant to the particular function or state of the program of interest, in order to study the program.

26. Use an editor or browser to traverse the code. Editors that support multiple windows can show several parts of the program at once.

27. File search tools such as UNIX `grep` can be used to find identifiers that may be in one of several files.

28. In the absence of tools like a cross-reference generator, such unlikely tools as spelling checkers can be useful for listing the identifiers used in the program.

29. Traditional debugging techniques can be used to *read* code. The addition of print statements, for example, can be useful in verifying hypotheses.

30. Read programs with a cross-reference listing, structure chart, or similar summaries of program information close at hand. It is sometimes useful to generate such charts by hand if they cannot be obtained automatically.

Although programmers commonly use compilers, editors, and prettyprinters, in the future, more sophisticated tools to assist with program reading may become available. Prototype tools for viewing programs in different ways and exposing dependencies among program components are described in [Cleveland89], [Pazel89], and [Wilde89]. It has even been suggested that stepwise abstraction may be automated to some degree. Wilde [Wilde90] discusses kinds of program dependencies that may be of interest to the program reader, as well as reading tools.

5. Teaching Program Reading

In this chapter, we consider activities that educators can use to teach students how to read programs, and help them to improve their program reading skills. We also consider how program reading can be used in situations where teaching reading skills is not the principal educational objective.

5.1. Need Program Reading Be Taught?

We argued in Chapter 2 that program reading skills are important to the software professional. But need we, as educators, work consciously to develop these skills among students, or may we assume that reading skills are somehow "picked up" along the way to gaining program writing competence?

Although it is no doubt true that some students seem to develop reading skills effortlessly—as effortlessly and mysteriously as they achieve writing skills—we believe that reading skills, in general, require cultivation. We have four reasons for this belief:

1. The nature of reading and writing skills.
2. The typical experiences of students.
3. Empirical evidence from published studies.
4. Our own classroom observations.

It should be obvious from our earlier discussion of comprehension studies that the activities one engages in when reading a program—recognizing, hypothesizing, verifying, abstracting—are different from the tasks normally associated with writing programs. There is no particular reason to expect these two sets of skills to be equally developed.

Students are probably called upon to read programs less frequently than they will be as software developers. Not only do they write programs more often than they analyze them, but also the programs they do write are seldom criticized in detail for style and documentation. As a result, student programmers develop personal styles that seem quite natural and correct, but which can be counterproductive when students are confronted with code written by someone else who is equally individualistic.

Empirical studies show that experienced programmers tend to be better program readers than novices are. Yet the studies also consistently show that reading strategies employed by experienced programmers are diverse, and usually do not correlate in any obvious way with experience [Crosby90, Littman86, Pennington87]. Experienced programmers often adopt disastrous reading strategies [Littman86, Pennington87]. Presumably, education can improve such performance, particularly because certain comprehension failures are systematic and, although they may not be totally preventable, perhaps we can arm students with defenses against them. (See [Letovsky86b].)

Our observations of student behavior suggest that students do not particularly like to read programs, possibly because doing so is difficult, possibly because it seems less creative or satisfying than writing code, certainly because students fail to grasp its impor-

tance. In any case, students often skim program examples in textbooks, rather than study them. Most students devote even less attention to sample programs distributed by teachers and often perform quite poorly on examination reading questions. (See [Deimel85c] for a simple example of this phenomenon.)

5.2. Teaching Strategies

Program reading is a complex, poorly understood cognitive activity. This surely makes it difficult to teach, but not, we think, impossible. In this section, we suggest what we believe to be useful teaching strategies for program reading. Although our remarks are meant to apply mostly to undergraduate education, where we believe program reading should first be taught, graduate educators and continuing education instructors also should be able find ideas here that they can use.

It is not our intention to presume that program reading should become the subject of a particular course. Instead, we believe that program reading should be discussed and practiced throughout the curriculum. Because reading programs is a skill, like writing programs, its mastery requires practice over an extended period.

A strategy designed to make students better program readers should have the following components:

1. Motivation: Students need to know that reading skills are important.
2. Theory: Students can be taught what is known about program comprehension and about particular reading strategies and techniques.
3. Demonstration: Teachers should not only lecture about reading techniques, but should also find opportunities to model reading behavior for their classes.
4. Practice: Students must be given opportunities to exercise program reading skills.
5. Reinforcement: Students must be encouraged, by feedback and grading, to use and improve their reading skills.

We need not say much about the first two components. Previous chapters have dealt with the importance of reading skills, our knowledge of program comprehension, and reading strategies and techniques. This material can be delivered through lectures, handouts, assigned readings, and offhand remarks in class. It is not necessary to spend a lot of class time on these topics, but it should be clear that the instructor considers them important. We believe that teaching about comprehension models, if only informally, is important early on because it helps students formulate what it is they need to know about a particular program, as well as the kind of knowledge they should be acquiring about programs generally.

Somehow, hearing about how to do something is less compelling than actually seeing the thing done. A teacher can demonstrate program reading by handing out a program to students and reading it himself from overhead slides, demonstrating what he does by thinking aloud. Since the skill he wants to demonstrate is program reading, not acting, the program used should be credibly unfamiliar. A variation is to videotape someone reading a program in this manner and show the videotape in class.

Student assignments are a good source of programs with which to demonstrate reading. A program from a newly collected assignment or one from a previous term can, after the programmer's name is removed, make excellent reading matter for a class. As a bonus, the teacher can give valuable feedback to the class concerning stylistic problems likely to be especially meaningful to the students. This can be an important opportunity, as time seldom permits giving such feedback with any regularity. Both "good" and "bad" programs can serve as examples. This will help students develop judgement and taste. Teaching assistants or graders can sometimes sort through programs to find suitable examples.

Another possible source of programs to read is system utilities, such as those available under UNIX. These programs can make good reading material, in part because students seem to like looking behind the veil and demystifying familiar programs usually regarded as primitives. The program in the Appendix is provided for reading, and the SEI has other source code available as well. Of course, the choice of reading material must be geared to the maturity of the students and the topics being covered in the course.

Reading programs in class offers an interesting alternative to conventional lectures. In advanced courses, in which students may be assumed to have at least modest programming skills, there is often little reason to "develop" a program on the blackboard to demonstrate, say, a new algorithm. Instead, the teacher can simply present a program, and then analyze it as would a professional programmer trying to understand any unfamiliar code. This procedure has the following advantages:

- It can save valuable class time.
- It demonstrates reading skills and reinforces their importance.
- It stresses the *analysis*, rather than *synthesis*, of programs (which is often the point of the lecture anyway).
- Since this can be done with code directly out of the textbook, it sends the message that "you, too, can make sense out of textbook examples."
- It relieves the instructor of the need to demonstrate his superiority by working out the program before the class off the top of his head.

There are many ways to give students practice reading programs, an essential part of improving their skills. Many instructors regularly distribute their own versions of program solutions to their classes after an assignment is due. These versions are usually intended to serve as models for program writing, rather than as material for reading practice. In principle, students should read these solutions and learn from them, perhaps finding better ways of performing operations they found troublesome when they were writing their own programs. Experience tells us, however, that students often simply file these programs away somewhere, never to be used, except possibly as tablecloths for late-night pizza. More incentive for reading is generally required. Here are possible ways of providing such incentive:

- Have students correct errors in their own programs after they are returned. Suggest they read the model program for ideas. This tactic benefits some students more than it does others. Students who have no errors to correct have no special incentive to look at the handout.

- Allow students a few days to study the sample program and resubmit their own, thus allowing them to improve their grades on the assignment. This, too, benefits students unequally, however.
- Have students add or remove functionality from the model program. They will already understand the application, and much of their efforts will be devoted to decoding the details of the model program. In fact, the model need not even be correct; students can be asked to find and remove a bug. Of course, if this is done outside of class, there is no way to prevent students from copying solutions from one another.
- Have students answer questions about the sample program. Although general questions (*e.g.*, "Write an essay explaining how this program performs its function.") can work, more limited and specific questions may be more effective at focusing students' attention.

Each of these strategies forces students to approach the program with particular objectives in mind. A related idea is to have students exchange programs and answer specific questions about them or evaluate the programs against specific criteria. This provides both reading practice and feedback on programming style and readability.

Reading exercises need not be associated with programming assignments, of course. At any time, the teacher can hand out a program listing or make available program files. This should be done in conjunction with a specific exercise or set of questions the students are asked to complete. Having students answer questions about programs is an excellent way to teach them programming techniques and increase their sophistication in thinking about programs. In fact, it can be a much more effective teaching method than classroom lecture, as the students become active participants rather than passive listeners.

Asking students to write essays about programs also provides an opportunity for students to sharpen their technical writing skills. (See the SEI curriculum module *Technical Writing for Software Engineers* [Levine90] for other suggestions to improve students' writing.)

It is not necessary that students write formal essays. Questions requiring brief but non-obvious answers can be quite effective in giving program reading practice and teaching whatever it is the instructor wants students to learn about programming. In fact, students should get experience handling narrow questions before being asked to deal with complex, open-ended ones. The latter category offers attractive possibilities, however. Students can be asked to compare alternative programs for the same task, or to evaluate a proposed—and possibly problematical—modification to a program.

It is possible even to have a little fun with programs distributed to the class. Try conducting a scavenger hunt with a program. Ask students to find as many odd features—not bugs—in a program as they can. Such features can include peculiar identifiers, unreachable statements, useless assignment statements, and so forth. Prizes can be offered to winners who find most oddities. Besides adding interest to the class, this activity emphasizes that even working programs are not necessarily perfect ones.

One of the authors has even offered a monetary bounty on any errors students could find—even typographical ones—in programs he distributed. This never proved costly, but perhaps the author was just lucky.

Other activities that might be tried with programs distributed to students include in-class code reviews and class discussions of programs students have been asked to study. Of course, code reviews should be part of any team project.

One of the authors regularly uses a type of assignment that combines program reading and program writing exercises. Students are given files for an incomplete program, which they are asked to finish. The program may include complete but undocumented procedures, procedures specified only by comments, and so forth. Students must read and understand what they have been given in order to complete the code and documentation.

Do not overlook the possibility of giving quizzes or examinations based on programs students have been asked to read outside of class. Students may be less effective preparing to answer unknown questions than ones given them in advance, however. On the other hand, this situation is realistic, as maintenance programmers are often given time to familiarize themselves with programs they are to maintain, even though they cannot know what, exactly, they will be called upon to do with them.

Perhaps a more attractive alternative for quizzes and examinations is the idea of using reading questions that do not depend on code studied in advance. Reading questions can be used to test reading skills, as well as program writing skills, or at least knowledge of programming techniques. (There is reason to believe there is some correlation between reading and writing skills. Experienced programmers, for example, seem to perform better in experiments involving program comprehension. See, for example, [Crosby90].)

Now we come to the matter of reinforcement. Students are usually willing to spend time learning a subject or performing a task only if there is some sort of tangible reward involved, usually in the form of a grade. This is a fact of academic life, and we can only accept it and use it to our advantage as educators. Students' grades *must* depend in part on reading activities if we are serious about imparting reading skills. Most of the activities we have mentioned can be graded in a straightforward way; they should indeed have grades associated with them. After a time, students will come to recognize improvement in their reading skills and their understanding of programming generally, and will, one hopes, engage in reading activities more enthusiastically.

Because becoming a good program reader is partly a matter of learning how to think about programs, instructors should explain what answers they were looking for in response to reading questions. Reviewing exercises in class can be a helpful way to give feedback, though students sometimes need personal comments as to why their answers were not correct.

Reinforcement for the idea that reading is important can also come from the use of programming style guidelines. If reading programs is important, so is writing more readable programs. Oman and Cook warn against specific guidelines, and recommend

teaching language-independent principles [Oman90b]. Teachers have to decide how much stylistic freedom to allow their students, however. In any case, style guidelines should be justified in terms of their implications for readability. Students can usually be convinced of the utility of reasonable, well-thought-out conventions, even if the conventions are not perfect. Upperclass students probably should be given more leeway to experiment with programming style, though they should be held accountable for their choices.

We are aware that most of the techniques suggested in this chapter require substantial work by the instructor. Writing programs for students to read, and devising follow-up projects, is not a simple matter. Because the benefits can be substantial, however, it may be worthwhile to establish a department-wide plan to create reusable code and program repositories where instructors can get (and place) software available for student reading exercises.

In the next chapter, we discuss construction of program reading questions. We show that a great deal of variety is possible and that questions need not be trivial. We also offer a framework helpful for generating good program reading questions.

5.3. Some Additional Ideas

We want to mention a few ideas related to program reading that may be helpful to educators, particularly those willing to experiment a bit with their courses.

We mentioned earlier the suggestion by Deimel and Moffat to restructure the introductory programming course by placing a heavy emphasis on reading. They recommend in [Deimel82] that a four-phase approach be used, in which students:

1. Use programs.
2. Read programs.
3. Modify programs.
4. Write programs.

The first phase introduces students to the kind of products they will later be asked to create. It helps them to develop ideas about what makes software desirable to users, independent of any understanding of how much work might be necessary on the part of developers to achieve an appropriate level of quality. In the second phase, students begin to learn a programming language. One hopes they develop intuition about how code should look, so that they make fewer "silly" composition mistakes later on. One also hopes they begin to see intelligent style conventions and internal comments in a positive light—as aids to comprehension, rather than as obnoxious requirements. The third phase lets students try their wings a bit and serves to emphasize the importance of maintaining software. Finally, students begin writing programs independently.

The Deimel/Moffat approach has a decided software engineering slant to it, and it may be an attractive strategy to try in undergraduate programs that have a serious commitment to software engineering. No thoroughgoing attempt to implement this approach —which would require a substantial body of materials—is known to the authors, however.

Although learning one's first programming language through program reading is something of an offbeat idea, programmers commonly learn a substantial amount about second and third languages by reading programs written in those languages. This idea suggests a number of educational possibilities, including ways to make language survey courses more meaningful. Project courses often require students to use a language with which they are unfamiliar. Providing students, in addition to the usual resources, with sample programs in that language and meaningful reading questions, such as those provided in Chapter 6, may ease the transition into use of the new language.

Finally, we suggest that encouraging students to do a *serious* read of their programs after they are "finished" can have a salutary effect on their work. Code reviews work, after all, because someone actually *looks* at the code, even if it appears to be running satisfactorily. Carefully and objectively reading one's program, even without the benefit of other reviewers, often uncovers previously unrecognized flaws. Students sensitized to the importance of program readability can also improve the readability of their own programs in this way. This is especially true with respect to internal comments. Knowledge of what the reader typically wants and needs to know, along with a detached assessment of whether or not the code succeeds at communicating that information, can lead to programs that are much better documented than they otherwise would be. Teachers skeptical that students will read their own programs may wish to provide opportunities for students to review one another's code.

We are at least a little embarrassed for our profession in suggesting that students should read what they write. No doubt an English composition teacher would find it unthinkable that a student should submit a theme without having read it. But programs seem to be different. Whereas a theme cannot be evaluated except by reading it, a program can be executed. And students seem disinclined to argue with the success of a properly running program. We should encourage them to behave otherwise.

6. Constructing Reading Exercises, with Examples

To teach program reading, it is essential to construct exercises that require students to answer questions about programs. In this chapter, we offer advice about framing individual questions and assembling groups of questions.

Our notion of a "program reading comprehension question" is broad. If a question or exercise requires understanding source code to answer it, that question fits our definition. In this sense, for example, a program maintenance exercise is a reading question. Program reading comprehension questions test reading ability and provide reading practice, which presumably builds reading skills. Questions must be *about* something, however, so that they necessarily also test for other skills and knowledge, a major reason to like reading questions, all other considerations aside. Rather than complicate matters unduly, we will generally speak of measuring program comprehension, ignoring any other objectives a question might achieve.

In Section 6.2, we list a number of questions based on the program in the Appendix. These questions are intended to be usable as they are, as well as to be suggestive of others that might be created by teachers, either for the program we have provided or for for other software.

6.1. Evaluating Program Reading Skills

A major problem of teaching program reading is that of evaluating students' understanding of the material. Program reading skills cannot be measured directly because the *product* of program understanding is in students' heads. As we shall see, however, a great variety of question types is available to us. Good comprehension questions can be constructed with just a little effort.

Much of the remainder of this chapter is based on [Deimel84], [Deimel85a], and [Deimel85c]. The reader should consult these papers for additional details and suggestions.

In general, we are interested in questions that require students to understand a program, procedure, or program fragment. The code may be presented on paper, it may be made available electronically, or it may even be executable. The code may contain meaningful comments or not. External documentation of various forms may or may not be available. We do not have space to explore these possibilities in detail. Clearly, however, certain choices make the same question easier or harder to answer. Information that is difficult to extract from raw source code can be very easy to find if provided directly in comments, for example. (More documentation is not always helpful, however.) When developing questions, it is important to ask what knowledge and skills we are trying to test. When evaluating a possible question, it is important to ask what knowledge and skills students will have to use (and therefore demonstrate) in order to answer it. These rules of thumb help us select questions and appropriate source materials.

For this report, we did what instructors often do as we developed our questions for an

exercise or examination, namely, made a few arbitrary decisions and took them as given. Thus, we wrote a well-documented Ada program and developed questions for a variety of educational objectives based on that program. The program can be provided on paper or electronically, in whole or in part. Readers can select sets of questions from those provided to test their students, or they can augment our questions with their own. If there is reason to do so, they can remove comments, change statements, or provide external documentation along with the source code.

Reading questions can be multiple-choice or free-response. One or more questions can be asked about a single passage. It is also possible to ask students to select one of several passages as the answer to a question. Possible forms of essentially the same question are illustrated in the examples below, which are adopted from [Deimel85a]:

1. What is the apparent purpose of the code below?

```
TEMP := A;
A := B;
B := TEMP;
```

2. What is the apparent purpose of the code below?

```
TEMP := A;
A := B;
B := TEMP;
```

   a. Set A, B, and TEMP all to the same value.
   b. Sort the values of A, B, and TEMP.
   c. Exchange the values of A and B.
   d. Sort the values of A and B.

3. Which one of the following fragments correctly and efficiently exchanges the values of A and B?

   a.
```
T1 := A;
T2 := B;
B := T2;
A := T1;
```

   b.
```
B := A;
A := B;
```

   c.
```
TEMP := A;
A := B;
B := TEMP;
```

   d.
```
T1 := A;
T2 := B;
A := T2;
B := T1;
```

The first question is a free-response question; the second and third are multiple-choice.

Multiple-choice questions have the advantage of being easy to grade and to grade objectively. Writing good multiple-choice questions can be time-consuming, however, a fact that may offset any scoring efficiency gained. Multiple-choice questions seem most desirable if a large pool of them can be developed over time. We have included a few multiple-choice questions among the exercises. Suggestions for making up such questions can be found in [Deimel85a], which also discusses the assessment of how successful particular questions are as evaluation instruments.

Various kinds of free-response questions are possible. Often, only a sentence or phrase can provide the answer. Essay questions, however, require more extended composition on the part students and expanded effort on the part of the instructor. Responses to questions can be diagrams or other kinds of documents. Cloze procedure tests have also been used for testing program comprehension. In a cloze procedure question, elements of the program are deleted (say, every *n*th statement or every *n*th operand or operator), and students are required to supply the deleted elements. A simple example might be:

1. Complete the fragment below, so that it exchanges the values of A and B:

```
TEMP := A;
_____________ ;
B := TEMP;
```

One of the most difficult aspects of generating reading comprehension questions is that of producing adequately diverse questions. The classification scheme of Deimel and Makoid [Deimel84] is an effective tool with which to deal with this problem. Deimel and Makoid suggest using a two-dimensional classification of question types. One axis represents behavioral objectives in the cognitive domain (roughly speaking, the cognitive difficulty of the operations required to produce the correct answer). The other axis represents the knowledge domain or domains (in Brooks's terminology) about which one must reason in order to answer the question. A set of questions representing question types well distributed in this two-dimensional matrix should, in principle, be a broad measure of reading comprehension. A set of questions that requires reasoning only about individual statements, on the other hand, would likely measure overall reading comprehension poorly.

The cognitive complexity axis uses Bloom's taxonomy of behavioral objectives in the cognitive domain [Bloom56, Bloom71]. This is a classification of types of behavior we might want to elicit from students. The taxonomy is hierarchical, at least in principle, in the sense that performance at any level requires the performance skills at all lower levels. The levels of the hierarchy are as follows (we offer brief explanations in the domain of our interest):

1. **Knowledge**: Knowledge-level tasks call for definitions, recognition, etc. The programmer is asked to respond with established facts about programs and programming methods, or to give back, more or less verbatim, what he has read.
2. **Comprehension**: This level represents simple understanding. The programmer is asked to summarize or paraphrase what he has read, but is not required to demonstrate deep understanding of it or its implication.

3. **Application**: This level involves the use of abstractions in particular and concrete situations. At this level, information becomes functional, not just theoretical. The programmer might be asked to describe the behavior of a program.
4. **Analysis**: This level is concerned with the organization of information and the relationships among elements. The programmer may be asked about program components, their organization, and how they work together.
5. **Synthesis**: Synthesis is the putting together of parts to form a whole. Writing a program or modifying a program are synthesis tasks.
6. **Evaluation**: The making of quantitative and qualitative judgements, requiring measurement against criteria. The programmer might be asked to evaluate program efficiency, readability, etc.

Deimel and Makoid in [Deimel84] and [Deimel85c] list several specific types of reading comprehension questions for each level. We will not reproduce the list here, but we discuss some of these types in relation to particular questions we provide in the next section.

6.2. Example Program Reading Questions

We now present concrete examples of program reading questions based on the multitasking Ada program provided in the Appendix. Our objectives are threefold:

1. To show that meaningful and varied reading questions are not difficult to construct.
2. To illustrate particular kinds of questions to help educators write questions matched to their own particular educational objectives.
3. To provide actual questions and associated source code that can be assembled by educators into exercises and examinations.

The questions are grouped by Bloom taxonomy level—roughly, in order of difficulty—though perhaps one should not take this description too literally. This arrangement facilitates both discussion of question construction and access by instructors, at least to the extent that they think in terms of abstract educational objectives. The list is not so long as to preclude a linear search for useful items. The questions are available on diskette (an order form is provided as an attachment), so searches of the question set can be made with a text editor.

We have not provided answers to questions. Although the reason for this has more to do with the publication schedule than with anything more philosophical, the net effect is not altogether bad. The lack of answers makes anyone who wants to use the questions look seriously at the accompanying code. This, in turn, helps one see the answers students will be asked to discover, appreciate the process students will need to go through to find them, and gain insight into the program—helpful for writing additional questions—and into program reading generally.

We should point out that these questions are not field tested. We encourage readers to create exercises or examinations from them and to share their materials and experiences with others through the Software Engineering Institute.

A few words are in order about the Ada program on which the questions are based. The program is a well-documented, multitasking program of about 800 statements. It is distributed among 11 files, whose combined length is about 3800 lines. The main procedure is `pdi`, which can be found in file *pdi.a*. The program searches (possibly very large) natural numbers in arbitrary bases to find what are called "perfect digital invariants" (PDIs) and "pluperfect digital invariants" (PPDIs). (The nature of these numbers is not important. The comments in the main procedure contain background information and references.) Significant features of the program include:

- A multiple-precision integer arithmetic package, `numbers`.
- A user-interaction capability that accepts user requests while searches are in progress.
- Automatic checkpointing and restarting, a useful feature, as searches can go on for hours or even weeks.

Most of the work of the depth-first search is carried out by recursive procedure `numbers.search.do_search.start_search.select_digits`. Although the program is not obviously implementation-dependent, its use of tasking and text_io may cause its operation to differ from one system to another when the program is run without modification. Reports of both problems and successes will be appreciated.

In the lists of questions beginning in Section 6.2.1, we have inserted space between items to allow the reader to make notes about individual questions. Questions at each level of the taxonomy are in separate sections. Below, we comment on a number of questions in order to give the reader a sense of what we have provided, to suggest kinds of questions that can be asked, and to show some of the thought process that underlies question construction. Note that the questions are not totally independent of one another, and use of one question may reasonably preclude use of some other one.

The knowledge-level questions in Section 6.2.1 require the most basic knowledge to answer. The questions shown can hardly be considered reading questions at all, relying as they do mostly on the knowledge students bring to them. It is difficult at this level to ask anything except basic Ada questions. Seldom, in fact, do we even want to ask knowledge-level questions in reading comprehension tests, except possibly in those for beginning students. Some items are arguably misclassified; question 1 is perhaps a comprehension-level question. The classification is not so much an end in itself, but a device for stimulating thought about reading questions, so there is no need to quibble about fine points. Question 5 involves the sort of knowledge that lends itself nicely to a multiple-choice question, as it is easy to generate candidate answers that differ slightly and systematically from one another. Well-prepared students have difficulty with such a question only to the extent that it takes a few moments to read the question and recognize where the correct answer is. Students who are less certain in their understanding are sometimes misled by distractors (wrong answers).

The comprehension-level questions in Section 6.2.2 require more understanding. Deimel and Makoid call questions like question 1 *interpretation* questions, in which the function of a statement or fragment is called for. Question 2 is also an interpretation type. Notice that it has been written in such a way as to elicit a response in the proper knowledge domain (in this case, the Ada statement domain). If a response were required in another domain (that of abstract data structures, for instance) this would probably not be a comprehension-level question. Care should be taken to cue students as to the kind of response being sought. In the absence of such a cue, students could answer this question in a more abstract domain, using abstraction as a cover for vagueness about just what is going on here. Questions 3 and 4 are *translation* questions, in which information needs to be transformed into another (in this case, graphic) form. Alternative questions might offer multiple diagrams, out of which students are asked to choose the most appropriate. Question 4 requires cognitive effort primarily in the data structures domain; question 3 concerns overall program architecture.

Question 1 of the application-level questions in Section 6.2.3 deals with the knowledge domains of numbers and operations on them (*i.e.*, the problem space) and that of procedures and functions. Question 1 requires a good deal of thought to answer, as alternative designs need to be worked out mentally. Good understanding of Ada visibility rules and private types is required to get the correct answer by any means other than guessing or magic. Questions such as this, which require students to relate different knowledge domains to one another, are probably somewhat more difficult to answer than comparable questions that do not do so. Question 3, a *prediction* question, is probably the sort of item most teachers would think of if asked to write a program reading question. The question can be answered by hand-tracing the code. Experienced programmers, however, would build an abstract model of the algorithm, from which the answer could be written down immediately, without step-by-step tracing. (It is a good idea to try to anticipate all possible methods of finding the answer to a question. It may be desirable to force or preclude use of a particular strategy.) Questions 5 and 6 are nearly identical, but they solicit different sorts of answers. The difference can be stated in terms of knowledge domains, but it may be more useful to think of the questions as asking “what?” and “how?”

Analysis-level questions are usually easy to write; they are often quite difficult to answer. Question 1 in Section 6.2.4 requires a clear understanding of both the problem domain and the algorithm used by the program. A complete answer to question 2 requires some specialized mathematical knowledge. (See the Deimel and Jones reference mentioned in the program comments.) Question 3 is a *justification* question—why did the programmer do what he did? (Question 6 deals with the same feature of the program.) Question 5 asks for an explanation; students must relate the problem domain terms in the question to lower-level program details, in order to give a correct explanation. Question 8, at least in part, is a *location*, or “where” question. Question 9 requires students to infer programmer style conventions (not a difficult task in this case). Questions 10 and 15 deal with Ada tasking. Note that answering most of the questions in this section require a good deal of abstract reasoning.

The synthesis-level questions in Section 6.2.5 all require students to construct something new or add new parts to an existing artifact. Questions 1, 2, and perhaps 11 are

particularly appropriate for a software engineering class. Most other questions involve program modifications. Questions 9 and 10 are unusual, in that they ask that functionality be *removed* from a program—sometimes a trickier job than adding functionality.

Evaluation-level questions are found in Section 6.2.6. Question 2 requires, if not formal methods, at least careful reasoning. Question 3 is a *criticism* question that makes students think carefully about some readability issues. Question 6, like question 1 in Section 6.2.3, requires firm understanding of Ada rules about packages. This question demands a good deal more clear thinking, however. This question allows for more than one answer to be correct. Such questions are a bit more complicated to grade than single-answer questions, but they sometimes seem irresistible.

6.2.1. Knowledge-Level Questions

1. Procedure `numbers."-".perform_subtraction` contains two loops involving `position`. Which one of the following statements is true?

    a. There is no declaration for `position`.

    b. There is no declaration for `position`, but there could be.

    c. The `position` in the first loop designates the same object as that in the second.

    d. The declaration of `position` occurs outside `numbers."-"`.

2. How many parameters has procedure `numbers."-".perform_subtraction`? What is the mode of each?

3. The line

```
SEPARATE (numbers.search)
```

    occurs several times in the program. What does this line mean?

4. The type `duration` appears several times in procedure `timekeeper.print_elapsed_time`. Which statement is true of `duration`?

    a. Type `duration` is a built-in type of the Ada language, whose complete semantics are described in the language reference manual.

    b. Type `duration` is imported from package `calendar`.

    c. Type `duration` is defined implicitly in package `timekeeper`.

    d. Type `duration` is defined in package `standard`.

5. What is the meaning of

```
WITH calendar;
```

at the beginning of package `timekeeper`?

a. Declarations in the body of package `calendar` are available within `timekeeper` without qualification.

b. Declarations in the specification of package `calendar` are available within `timekeeper` without qualification.

c. Declarations in the body of package `calendar` are available within `timekeeper`, provided they are qualified.

d. Declarations in the specification of package `calendar` are available within `timekeeper`, provided they are qualified.

6.2.2. Comprehension-Level Questions

1. What is the purpose of the statement

```
FOR do_search_type'storage_size USE 1_048_576;
```

   in the specification of package `numbers.search`?

   a. It specifies how much memory to allocate.
   b. It specifies the address at which part of the object code is to be loaded.
   c. It allows certain objects to be used without a qualifying identifier.
   d. Using iteration, it performs initialization.

2. As literally as possible, explain what the two statements inside the loop in function `numbers.convert_to_string` do. Be sure to discuss the use of the " ' " notation.

3. Draw a Booch diagram showing the relations among the main procedure, tasks, and packages of the program.

4. Draw one or more diagrams to show in detail the structure of `sum_table` in `numbers.search.do_search.start_search`.

5. List all the operators the programmer has overloaded in this program.

6.2.3. Application-Level Questions

1. Let package `numbers` be used in another program by means of a `WITH` clause. Suppose it is necessary to multiply two 25-digit, base-19 integers together in a reasonably efficient manner, taking advantage of the operations defined in the specification of `numbers`. Which one of the following is true?

    a. The operation can be performed directly by `numbers."*"`.

    b. The multiplication could be performed if a function were provided to extract the *n*th digit of a multiple-precision integer.

    c. The required operation is impossible to perform in the absence of the assignment operator for multiple-precision integers.

    d. The operation is easily performed using `numbers."*"` and `numbers."+"`.

2. In function `numbers."-"` (binary minus), indicate which instances of operators are overloaded by the programmer.

3. Assume `numbers.search.do_search.start_search.initialize_table` is called with parameters 4 and 3. List the numbers (*i.e.*, the multiple-precision integer values) stored in the elements of `sum_table` in the order generated. Be sure to identify which value is assigned to which element of `sum_table`.

4. Give a valid compilation order for the 11 program files.

5. What happens when the user requests that input be taken from the normal input file (*i.e.*, not the checkpoint file or default input file) but that file does not exist? State clearly what the program does, as seen by the user.

6. What happens when the user requests that input be taken from the normal input file (*i.e.*, not the checkpoint file or default input file) but that file does not exist? State clearly what the program does and explain why it does it.

6.2.4. Analysis-Level Questions

1. The program appears to find PDIs in descending order. If it always does so, explain why this is so. If not, explain why this might *seem* to be true.

2. What *exactly* is the significance of the number output when the program displays "Number of combinations tested"? Is this number helpful to the user in determining how much of a search has been performed? How difficult would it be to display a number that told, of the theoretically possible numbers that might be PDIs, how many of them have definitely been eliminated as PDI candidates?

3. In task `numbers.search.do_search`, two different data structures are used to hold information about the combination of digits that has been selected. Why?

4. The data type `dynamic_string` appears a number of times in package `numbers`. What is it, and how is it used?

5. How does the program avoid testing every positive integer of the desired length for the property of being a PDI?

6. What is the significance of the loop

```
FOR digit IN numeral LOOP
   wide_select_vector(digit) := select_vector(digit);
END LOOP;
```

in procedure `numbers.search.do_search.start_search.record_checkpoint`? That is, why is this loop necessary?

7. What change to the program probably should result in a change of the 2 in procedure `numbers.search.do_search.start_search.print_state` to some other value?

   a. A change in the value of `numbers.maximum_radix` to 110.

   b. A change in the value of `numbers.maximum_digit_length` to 1.

   c. A change in the value of `numbers.maximum_number_length` to 105.

   d. A change in the value of `numbers.maximum_radix` to 9.

   e. None of the above.

8. What limitations are there on the PDI searches the program is able to perform? Where in the code are these limitations established?

9. Describe the programmer's convention with respect to the use of upper- and lowercase.

10. Under what circumstances, if any, can more than one task be making progress at the same time? (Waiting for an entry call is *not* considered making progress.)

11. How does the program know what checkpoint file to read? How does it know what checkpoint record to read?

12. What are all the circumstances under which function `numbers."+"` returns a value flagged as erroneous? Can any such circumstances actually occur in this program?

13. In procedure `numbers.search.do_search.start_search`, two functions from package `timekeeper` are renamed. Where, if at all, are these functions actually used?

14. Explain the operation of procedure `numbers.search.do_search.start_search` in terms of tasks, procedures, and functions.

15. Describe the overall operation of the program in terms of interacting tasks.

16. Explain the method used by the program to restart from a checkpoint, that is, to reproduce a program state equivalent to the state of the program at the time the checkpoint was written. The explanation should be general enough to allow the reader to use the method in another, unrelated program.

6.2.5. Synthesis-Level Questions

1. Write a test plan to determine how well the program handles all errors associated with files *input.dat* and *pdi.ckp*.

2. Write a test plan to exercise all exception handlers in the program related to operations on files *input.dat* and *pdi.ckp*.

3. What is the significance of the literal 2 in procedure `numbers.search.do_search.start_search.print_state`? Replace this literal with an appropriately named and declared constant.

4. Add a feature to the program to allow search parameters to be taken from the default input file (presumably the keyboard), in addition to the options currently offered.

5. Rewrite function `numbers."+"` with the intention of making it as time-efficient as possible.

6. Presumably, procedure `numbers.search.do_search.start_search.-select_digits` would be faster if implemented without recursion. Propose a design for such an iterative procedure. Describe in detail any necessary data structures and their use.

7. Presumably, procedure `numbers.search.do_search.start_search.-select_digits` would be faster if implemented without recursion. Implement a design that eliminates the use of recursion.

8. Change the program to accommodate different line widths in the default output file. In the modification, it should be easy to change the line width associated with the program.

9. In order to increase the efficiency of searches, the program could be modified so as to disable the user-interaction feature while a search is in progress. Describe approaches to doing this, assuming the code is to be changed as little as possible.

10. In order to increase the efficiency of searches, completely remove the user-interaction capability that allows the user to interact with the program while a search is being carried on. Be sure to change all relevant comments in the code.

11. Briefly summarize the programmer's indentation conventions in a style guide you could give to another programmer.

6.2.6. Evaluation-Level Questions

1. Although the user's ability to interact with the program while a search is being carried on is useful, it presumably creates some execution overhead. Explain the nature of this overhead and what, if anything, the user can do to manage it.

2. Prove (possibly informally) that function `numbers."<="` performs correctly.

3. Explain the programmer's use (and non-use) of blank lines in the declarations at the beginning of the specification of package `numbers`. Does this use enhance readability of the declarations? If possible, suggest a rule about the use of blank lines (or related formatting conventions) that would improve readability of these declarations.

4. In what way is the manner in which the programmer has written Boolean expressions unusual? Why do you think the programmer did what he did? Do you agree with his reasoning?

5. Function `numbers."-"` (binary minus) appears to have been written from `numbers."+"`, or vice versa. Support or refute this possibility. Are the two implementations equally appropriate for their respective tasks?

6. Arguably, package `search` is not most logically placed as part of package `numbers`. Which of the following statements are true or probably true?

    a. Package `search` is not syntactically necessary.

    b. Task `search.do_search` must be defined within package `numbers` because an Ada task cannot stand alone.

    c. Task `search.do_search` is defined within package `numbers` primarily so that identifiers do not have to be qualified.

    d. Package `search` is defined within package `numbers` because both are needed for conducting searches for PDIs and are unlikely to be useful in any other context.

    e. If the package `search` were defined outside of package `numbers`, the inability to assign multiple-precision integers would complicate the programming within `search`.

Annotated Bibliography

Baecker90

Baecker, Ronald M., and Aaron Marcus. *Human Factors and Typography for More Readable Programs*. Reading, Mass.: Addison-Wesley, 1990.

The authors argue that the emphasis on writing, rather than on reading programs has caused the *visual* (as opposed to *logical*) aspects of programs to be ignored. Attempts to program in a more visual way (so-called "visual programming") have not been successful, so the authors were led to examine *program visualization*. In particular, they have applied graphics design principles and techniques to the visual display of C programs. The result is enlightening and striking. Their programs use multiple typefaces and font sizes, elaborate spacing conventions, shading, color, and the like. Similar treatments could be given programs written in other languages. The authors justify their design choices and provide useful background information on program comprehension studies and the elements of typography.

This book is similar in spirit to the work of Oman and Cook [Oman90a, Oman90b], though it is more radical in its suggestions. The book is most helpful in suggesting possibilities likely to be overlooked because we have accepted the technological limitations of the past as givens. Persistence and familiarity with C are prerequisites to getting through this substantial book.

Basili82

Basili, Victor R., and Harlan D. Mills. "Understanding and Documenting Programs." *IEEE Trans. Software Eng. SE-8*, 3 (May 1982), 270-283.

Abstract: *This paper reports on an experiment in trying to understand an unfamiliar program of some complexity and to record the authors' understanding of it. The goal was to simulate a practicing programmer in a program maintenance environment using the techniques of program design adapted to program understanding and documentation; that is, given a program, a specification and correctness proof were developed for the program. The approach points out the value of correctness proof ideas in guiding the discovery process. Toward this end, a variety of techniques were used: direct cognition for smaller parts, discovering and verifying loop invariants for larger program parts, and functions determined by additional analysis for larger program parts. An indeterminate bounded variable was introduced into the program documentation to summarize the effect of several program variables and simplify the proof of correctness.*

The authors take a modest FORTRAN subroutine for finding roots of a function (ZEROIN, which is fewer than 150 lines long including comments) and reverse engineer it to produce a specification and correctness proof, documentation sufficient to answer several questions posed about the code and, presumably, adequate to meet future maintenance needs. This approach is consistent with the authors' belief that code and a correctness proof should be developed from a specification in the first place.

The code is first restructured into one composed of prime programs (see [Linger79]), a process requiring the duplication of several lines of code of the unstructured original. Hypotheses are generated about the resulting components, which are then confirmed by proving theorems. A table of data references is used to provide preliminary insights into the semantics of the program. Results of individual theorems are combined until statements can be proved about the entire subroutine.

This paper demonstrates something of a "brute force" approach to determining what a program does and writing down the findings. This is more important than it sounds; the goal of generating a correctness proof provides structure to the discovery process of learn-

ing what the program does. The strategy this imposes on the reader is certainly used by programmers, though in limited circumstances and with less formality. The authors are more prescriptive than descriptive, however, and are not suggesting that programmers actually follow their procedure.

It is too bad the authors offer only pages and pages of proofs as documentation and do not suggest what comments they might actually insert into the restructured code. It should be kept in mind, too, that the techniques demonstrated cannot capture certain pragmatic information about how the program was intended to be used if this information is not explicitly represented in the program.

Teachers and students should read this paper. It demonstrates well how we can reason about code and is likely to provoke heated discussion about how much work is enough when reading programs.

Bentley86a

Bently, Jon, and Donald E. Knuth. "Literate Programming." *Comm. ACM 29*, 5 (May 1986), 364-369. One of Bentley's "Programming Pearls" columns.

A nice description of Knuth's WEB system and his scheme of "literate programming." The idea here is to free programming from the arbitrary restrictions of programming languages and to produce essay-like texts for human consumption. A source file is prepared with ordinary English text, programming language statements—Knuth's system is Pascal-based—and imbedded TEX formatting commands. Compilers WEAVE and TANGLE produce, on the one hand, a human-readable program with commentary and, on the other, a thoroughly unreadable program for machine use. The system allows the programmer to introduce and annotate programming language statements in logical order, rather than some order dictated by a language standard. A two-page example program is included as a sidebar, and it communicates well the spirit of Knuth's programming method.

No programmer should miss this column. Even if literate programming does not seem to you the wave of the future, Knuth's style of exposition is exceedingly thought-provoking. (See also [Bentley86b].)

Bentley86b

Bently, Jon, Donald E. Knuth, and Douglas McIlroy. "A Literate Program." *Comm. ACM 29*, 6 (June 1986), 471-483. Another "Programming Pearls" column.

This column is a follow-up to [Bentley86a]. It contains a nearly 8-page literate program by Knuth, followed by a review (literary criticism, if you will) by Doug McIlroy. This is not essential reading, but it is very interesting—for its literate program, programming criticism, and use of a novel data structure.

Bloom56

Bloom, Benjamin S., *et al.*, eds. *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*. New York: David McKay, 1956.

Source of Bloom taxonomy of educational objectives in the cognitive domain.

Bloom71

Bloom, Benjamin S., John T. Hastings, and George F. Madaus. *Handbook of Formative and Summative Evaluation of Student Learning*. New York: McGraw-Hill, 1971.

Provides additional information on the Bloom taxonomy.

Brooks78

Brooks, Ruven. "Using a Behavioral Theory of Program Comprehension in Software Engineering." *Proc. 3rd Int. Conf. on Software Eng.* New York: IEEE, 1978, 196-201.

Abstract: *A theory is presented of how a programmer goes about understanding a program. The theory is based on a representation of knowledge about programs as a succession of knowledge domains which bridge between the problem domain and the executing program. A hypothesis and verify process is used by programmers to reconstruct these domains when they seek to understand a program.*

The theory is useful in several ways in software engineering: It makes accurate predictions about the effectiveness of documentation; it can be used to systematically evaluate and critique other claims about documentation, and it may even be a useful guideline to a programmer in actually constructing documentation.

In this paper, Brooks sets forth a model of program comprehension and relates it to the nature of programs and their documentation. His basic ideas reappear in [Brooks82] and in [Brooks83], where the *process* of comprehending a program receives greater emphasis. The author argues strongly against the notion put forth by Kernighan and Plauger that "the only reliable documentation of a computer program is the code itself" [Kernighan74]. He offers intuitively appealing arguments for his theory, as well a very brief description of a supporting experiment.

This paper is easy reading and likely to elicit a few "ahas" from anyone who takes programming seriously. The most assailable part of Brooks's theory, his severely top-down description of how people actually read programs, receives relatively little attention here.

Brooks82

Brooks, Ruven. "A Theoretical Analysis of the Role of Documentation in the Comprehension of Computer Programs." *Proc. Conf. on Human Factors in Computer Systems.* New York: ACM, 1982, 125-129.

Brooks describes his theory of program comprehension and uses that theory to draw inferences about documentation. Brooks's most striking conclusion is that it is as important to document the problem domain as it is to document the program itself. He concludes that different programming languages require different kinds of documentation, and he asserts that multiple forms of documentation are beneficial when they convey different kinds of information.

The description of Brooks's theory is better developed in [Brooks83], although this paper serves as a brief, readable introduction to it. This paper demonstrates the utility of the theory, although readers may find themselves wanting more details. Teachers may want to ask students to build on the inferences presented in the paper.

Brooks83

Brooks, Ruven. "Towards a Theory of the Comprehension of Computer Programs." *Intl. J. Man-Machine Studies 18*, 6 (June 1983), 543-554.

Abstract: *A sufficiency theory is presented of the process by which a computer programmer attempts to comprehend a program. The theory is intended to explain four sources of variation in behavior on this task: the kind of computation the program performs, the intrinsic properties of the program text, such as language and documentation, the reason for which the documentation is needed, and differences among the individuals performing the task. The starting point for the theory is an analysis of the structure of the knowledge required when a program is comprehended which views the knowledge as being organized into distinct domains which bridge between the original problem and the final program. The pro-*

gram comprehension process is one of reconstructing knowledge about these domains and the relationship among them. This reconstruction process is theorized to be a top-down, hypothesis driven one in which an initially vague and general hypothesis is refined and elaborated based on information extracted from the program text and other documentation.

This extended treatment of Brooks's ideas about program comprehension is intended to provide an adequate descriptive model of how programmers understand programs. According to this model, when one understands a program, one has constructed a mental model of successive knowledge domains bridging from the problem domain to the domain of the program in execution. Each of these domains consists of objects, properties, relations, and operations. The succession of domains may include the problem domain, the domain of a mathematical model of the problem, the algorithm domain, the programming language domain, etc. One must also understand the relationships that exist between adjacent domains. The process of reading a program to understand it is one of constructing a model of this sort or, depending upon one's reading objectives, constructing a part of one. According to Brooks, this process is largely top-down: the reader generates hypotheses about the program, which he then attempts to verify from the code and whatever other documentation is available. This verification task is aided by "beacons," code features characteristic of recurring structures or operations. Hypotheses are generated hierarchically until hypotheses can be bound to particular code segments. In this process, hypotheses are frequently revised or replaced by more credible ones.

The paper concludes with a discussion of factors affecting comprehension, including the nature of the problem, available documentation, programmer knowledge (both of programming and of the problem domain), reading goals, and reading strategy.

Although not overtly based on empirical studies, Brooks's model seems both sensible and serviceable. The model is somewhat dogmatic about programs being read top-down, however; and although the author acknowledges the bottom-up strategies illustrated in [Basili82] and elsewhere, he dismisses them as less powerful and less important.

This is an important, well-written paper. It is particularly concerned with the *process* of comprehension, however, and the reader wishing to better understand what it means to understand a program should look at [Brooks78]. It may be worthwhile to ask students if they believe people actually read programs as Brooks describes.

Cleveland89

Cleveland, L. "A Program Understanding Support Environment." *IBM Systems J. 28*, 2 (1989), 324-344.

Abstract: *Software maintenance represents the largest cost element in the life of a software system, and the process of understanding the software utilizes 50 percent of the time spent on software maintenance. Thus there is a need for tools to aid the program understanding task. The tool described in this paper—Program UNderstanding Support environment (PUNS)—provides the needed environment. Here the program understanding task is supported with multiple views of the program and a simple strategy for moving between views and exploring a particular view in depth. PUNS consists of a repository component that loads and manages a repository of information about the program to be understood and a user interface component that presents the information in the repository, utilizing graphics to emphasize the relationships and allowing the user to move among the pieces of information quickly and easily.*

This paper is a thorough description of the PUNS system developed at IBM. PUNS exists as a research prototype for IBM System/370 Assembly Language. The PUNS repository resides on an System/370 30XX computer, while the user interface can reside on a workstation running Microsoft Windows. PUNS supports the program understanding task by organizing and presenting the program from many different viewpoints: a call graph for a

collection of procedures, a control flow graph for a single procedure, a graph that relates a file to the procedures that use it, a data flow graph, a use-definition chain for a variable. The tool uses static analysis techniques to detect low-level relationships that exist within the program. It consolidates and organizes these relationships and presents them in a user-friendly environment. Prior to using PUNS, the user is required to structure a database for the particular program to be investigated.

Several extensions are under way to make PUNS a more useful tool. Perhaps the most important one reported in the paper is called a dynamic information updating facility. As it stands now, PUNS requires the repository to be established before a user session can begin. However, there is much to be gained by allowing the user to update information during the session, since there are relationships that cannot be determined using the static analysis employed when the repository is set up. Of course, as is discussed in the paper, allowing for dynamic additions to the repository raises the question of accuracy of information retrieved from the system. Attempts to extend PUNS to operate on higher-level languages are not reported.

PUNS provides the user with a powerful tool to aid in understanding a program. Although it is limited to assembly language programs, it suggests the kind of tool that may one day become commonplace for use with high-level languages.

Corbi89

Corbi, T. A. "Program Understanding: Challenge for the 1990's." *IBM Systems J. 28,* 2 (1989), 294-306.

Abstract: *In the Program Understanding Project at IBM's Research Division, work began in late 1986 on tools which could help programmers in two key areas: static analysis (reading the code) and dynamic analysis (running the code). The work is reported in the companion papers by Cleveland [Cleveland89] and by Pazel [Pazel89] in this issue. The history and background which motivated and which led to the start of this research on tools to assist programmers in understanding code is reported here.*

The author makes a case for the study and development of program reading skills, both in the workplace and in the classroom. The paper quotes a large number of authors who, for the last two decades, have made predictions (some true, others not quite so) on software development and maintenance. Certainly aging software systems are an integral part of today's (and tomorrow's) software. Corbi stresses the point that program readers will be needed in the 1990s, but that, unfortunately, program reading instruction is missing from most computer science curricula. The paper suggests approaches to maintenance of existing systems and discusses current tools and their limitations. Finally, the author gives his own view on software maintenance for the '90s.

This paper is recommended as a reading assignment for upperclass undergraduates. Furthermore, anyone interested in program reading, both as a necessary professional skill and as an activity within computer-related curricula, will find this paper most helpful. Instructors can find plenty of motivational statements to share with colleagues and students alike. The paper contains a substantial bibliography.

Crosby90

Crosby, Martha E., and Jan Stelovsky. "How Do We Read Algorithms? A Case Study." *Computer 23,* 1 (Jan. 1990), 24-35.

The authors report on a study that examined eye movements of programmers reading a Pascal version of a binary search, as well as their eye movements while studying slides illustrating the operation of the algorithm. Nineteen low- and high-experience programmers, all but one a student, served as subjects. Subjects were given a pretest on the binary

search, and cloze tests were used to measure comprehension. Fixation times and number of fixations were gathered. Most of the results reported are qualitative, however, and the "experiment" seems not to have addressed any particular hypothesis.

The authors claim support for the "immediacy theory" (that text is processed immediately, as opposed to being stored in a mental buffer for later cognitive processing), but they report no evidence for major systematic differences in reading strategy between novices and experts. High-experience subjects devoted more attention to "meaningful areas" of the code, however. Reading strategies differed in the relative attention given to comments and code, in the number of passes over the text, and in the degree to which subjects compared program elements to one another rather than reading left-to-right and top-to-bottom.

This paper is most interesting for its graphic analysis of the data to discover patterns in reading strategies. Although generalizations valid for *really* experienced programmers and for realistic blocks of code are not readily apparent, the paper nonetheless suggests interesting future work.

The paper is easy reading, though it is somewhat vague about details of the experimental protocol.

Davis84

Davis, John S. "Chunks: A Basis for Complexity Measurement." *Info. Processing & Mgt. 20*, 1-2 (1984), 119-127.

Abstract: *The state of the art in psychological complexity measurement is currently at the same stage as weather forecasting was when early Europeans based their predictions on portents of change. Current direct measures of program characteristics such as operator and operand counts and control flow paths are not based on convincing indicators of complexity. This paper provides justification for using chunks as a basis for improved complexity measurement, describes approaches to identifying chunks, and proposes a chunk-based complexity measure.*

This paper focuses more on the uses of the abstraction operation of "chunking" as a means of measuring program complexity than on how to extract chunks from programs. The chunks Davis is concerned with are at the level of Letovsky's and Soloway's "plans" [Letovsky86b].

Some of the earliest studies of chunking examined chess players. Experiments show that master players can remember more than novices can from a quick scan of a chess board if the chess board represents a meaningful situation. If, however, the pieces are randomly arranged, non-master players do as well as chess masters. In the programming world, chunks can be thought of as patterns of statements that accomplish a particular task. For example, experienced programmers may recognize the familiar pattern of a sorting algorithm. Davis reports that the Raytheon Company found that about half the code in its inventory of COBOL programs was "redundant," in the sense that similar code existed to perform essentially the same function.

The paper proposes two chunk-based complexity measurement models and reports on comprehension experiments aimed at validating proposed metrics. Davis points out that programmers often maintain the same piece of code over a long period of time. Comprehension experiments that present subjects with unfamiliar programs may therefore be less relevant to the maintenance task than it might at first appear.

The paper is recommended for students with some level of programming maturity. The results reported could be the basis of interesting class discussion in an advanced course.

Deimel82
Deimel, Lionel E., and David V. Moffat. "A More Analytical Approach to Teaching the Introductory Programming Course." *Proc. Natl. Educational Computing Conf. 1982.* Columbia, Mo.: University of Missouri, 1982, 114-118.

The authors review approaches to teaching the introductory programming course and conclude there is a need for a radically different approach, largely because students fail to grasp the nature of programs and of program development. Their solution is more *analytical* (as opposed to *synthetic*), consisting of four stages of instruction, in which students successively (1) experience programs as a user, (2) read and analyze programs and algorithms, (3) modify existing programs, and (4) design and implement new programs. Much of the discussion is concerned with the benefits of program reading.

This is a paper for teachers. The authors acknowledge that the lack of appropriate materials makes their approach difficult to implement.

Deimel84
Deimel, Lionel E., and Lois Makoid. "Measuring Program Reading Comprehension: The Search for Methods." *NECC '84: 6th Annual Natl. Educational Computing Conf.* Dayton, Ohio: University of Dayton, 1984, 142-146.

Abstract: *Evaluating program reading comprehension is a difficult task faced by both programming instructors and software psychologists. This paper offers a taxonomy of methods to measure comprehension and relates these techniques to a theory of comprehension. The classification should be useful for constructing test questions for both the classroom and laboratory.*

Deimel and Makoid present their classification of program reading comprehension questions that we have used in this report. The two-dimensional taxonomy is based on the Bloom taxonomy of educational objectives in the cognitive domain [Bloom56, Bloom71], and on Ruven Brooks's knowledge domains [Brooks83]. The authors present a list of question types by Bloom taxonomy level, a list refined in [Deimel85c].

This paper is primarily for teachers, though it might be of interest to students needing to construct comprehension tests in experimental studies.

Deimel85a
Deimel, Lionel E., and Lois Makoid. "Developing Program Reading Comprehension Tests for the Computer Science Classroom." *Computers in Education: Proc. IFIP TC 3 4th World Conf. on Computers in Education—WCEE 85, Norfolk, VA, USA, July 29-August 2, 1985.* Amsterdam: North-Holland, 1985, 535-540.

Abstract: *A methodology for constructing program reading comprehension tests is discussed and illustrated. Emphasis is on multiple-choice tests used with realistic reading passages. Item writing employing a classification of question types developed by the authors and a program comprehension model developed by Ruven Brooks is recommended.*

The authors discuss constructing multiple-choice program reading comprehension tests within the framework described in [Deimel84]. The paper is illustrated with examples and provides a good deal of practical advice to computer science instructors unfamiliar with education literature. Most of the examples are taken from a reading passage and comprehension test reproduced completely in [Deimel85b].

This paper and the next are addressed to teachers, and may not be of much interest to students.

Deimel85b

Deimel, Lionel E., Lynda Kunz, Lois Makoid, and Jo Perry. "The Effects of Comment Placement and Reading Times on Program Reading Comprehension." *Proc. 19th Ann. Conf. on Info. Sciences & Systems.* Baltimore: The Johns Hopkins Univ. Dept. of Electrical Eng. & Comp. Sci., 1985, 595-601.

Abstract: *Two experiments are described which compare the use of detailed comments placed either in line with the high-level language code or offset to the right. Given more than adequate time in Experiment 1 to read the program and answer comprehension questions, subjects given the two commenting treatments scored similarly on a comprehension test. When reading time was substantially reduced in Experiment 2, there were significant differences in comprehension between the commenting styles, favoring offset commenting. Further analysis revealed a significant time by comment placement interaction. Possible explanations and related questions are discussed.*

The commenting styles discussed and their possible effects on program readability are certainly of interest here, but perhaps of greater interest is the construction of the test by which program comprehension was measured. The test was constructed using the theory set forth in [Deimel84]. Most of the experimental materials are reproduced in the paper.

Teachers or experimenters needing to construct program comprehension tests will find this paper quite interesting.

Deimel85c

Deimel, Lionel E. "The Uses of Program Reading." *ACM SIGCSE Bulletin 17*, 2 (June 1985), 5-14.

Abstract: *It is argued that program reading is an important programmer activity and that reading skill should be taught in programming courses. Possible teaching methods are suggested. The uses of program reading in test construction and as part of an overall teaching strategy is discussed. A classification of reading comprehension testing methods is provided in an appendix.*

This paper argues for the importance of program reading and contends that reading skills are not necessarily developed in students unless the students receive explicit instruction designed to develop these skills. The author contends that three components are needed to teach program reading—lecture, reading exercises, and program writing standards that are designed with the production of comprehensible programs in mind. He also suggests that program reading can be used both to teach and to evaluate general programming skills.

The appendix contains a revised version of the question classification introduced in [Deimel84]. This list is an improvement over the earlier one, although there are omissions and, perhaps, some misclassification.

This paper is primarily addressed to teachers.

Goldberg87

Goldberg, Adele. "Programmer as Reader." *IEEE Software 4*, 5 (Sept. 1987), 62-70. Paper originally appeared in *Information Processing 86: Proc. IFIP 10th World Comp. Conf.*, H. J. Kugler, ed. Amsterdam: North-Holland, 1986, 379-386.

This paper describes how the facilities of the Smalltalk-80 environment support program reading, a particularly important function in what the author calls an "exploratory environment," in which much programming is accomplished by modifying and reusing existing application and system code. Goldberg describes the Smalltalk-80 system in terms

of four levels (user interface, functionality, structure, and language/implementation) and lists important comprehension questions for each level.

This paper offers an unusual argument for needing to read programs, an argument tied quite directly to program writing. It suggests ways in which future environments may provide support for both activities.

Kernighan74
Kernighan, Brian W., and P. J. Plauger. *The Elements of Programming Style.* New York: McGraw-Hill, 1974. A second edition was published in 1978.

This influential book tries to do for programming what Strunk and White did for writing. The authors want people to read programs and thereby learn to program better. This slim volume is filled with snippets of advice ("*Make your programs read from top to bottom*.") and illustrative code segments from a variety of languages. It is not the ultimate authority some would make it out to be, but it is a stimulating classic that everyone interested in serious programming should read. It is somewhat dated, but contains a lot of good advice.

Kernighan81
Kernighan, Brian W., and P. J. Plauger. *Software Tools in Pascal.* Reading, Mass.: Addison-Wesley, 1981.

In *The Elements of Programming Style*, Kernighan and Plauger illustrate their points with other people's code. Here, they use their own simple, reusable software tools, written in Pascal. The book is virtually a whole course on programming technique. There is much code to read here, but, as is commonly the case in textbooks, most of it is inscrutable without the surrounding discussion. Thousands of programmers have studied this book on their own. (This is a revision of an earlier book, *Software Tools*. The programs in that book are written in Ratfor, which requires a preprocessor whose output is FORTRAN code.)

Knuth84
Knuth, Donald E. "Literate Programming." *Computer J. 27*, 2 (May 1984), 97-111.

Knuth describes his WEB system for programming and documentation. Anyone with a deep interest in the system should read this paper, but the reader who would prefer a brief, lucid description of this interesting system (and philosophy) should read Jon Bentley's piece [Bentley86a] on the subject instead.

Knuth86a
Knuth, Donald E. *METAFONT: The Program*. Reading, Mass.: Addison-Wesley, 1986.

Source code for Knuth's typeface-generation system, written using WEB. (See [Bentley86a].)

Knuth86b
Knuth, Donald E. *TEX: The Program*. Reading, Mass.: Addison-Wesley, 1986.

Source code for Knuth's text-processing system, written using WEB. The book runs to nearly 600 pages. (See [Bentley86a].)

Letovsky86a
Letovsky, Stanley. "Cognitive Processes in Program Comprehension." In *Empirical Studies of Programmers: Papers Presented at the First Workshop on Empirical Studies of Programmers, June 5-6, 1986, Washington, D.C.*, Elliot Soloway and Sitharama

Iyengar, eds. Norwood, N.J.: Ablex, 1986, 58-79. Reprinted in *J. Syst. and Software 7*, 4 (Dec. 1987), 325-339.

Abstract: *This paper reports on an empirical study of the cognitive processes involved in program comprehension. Verbal protocols were gathered from professional programmers as they were engaged in a program understanding task. Based on analysis of these protocols, several types of interesting cognitive events were identified. These include asking questions and conjecturing facts about the code. We describe these event types, and use them to derive a computational model of the programmers' mental processes.*

Letovsky refers to a study involving the videotaping of six professional programmers as they enhanced a FORTRAN 77 program of about 250 lines. (The same study is also the basis for [Letovsky86b] and [Littman86].) Subjects were asked to "think aloud" as they worked. The author describes and analyzes what they said as they labored to understand the program to be modified. He presents a cognitive model of program understanding composed of the programmer's *knowledge base*, a *mental model*, the construction of which is the ultimate goal of program reading, and an *assimilation process* by which the programmer actually builds the mental model. Most of the paper is concerned with the assimilation process and the empirical data justifying the author's analysis of it.

Although Letovsky's language often differs from that of Brooks, his cognitive model of program comprehension is basically consistent with and elaborates the model in [Brooks83]. Whereas Brooks emphasizes top-down approaches to reading programs, Letovsky offers convincing evidence that programmers work both top-down and bottom-up. Much of the paper is devoted to analysis of the "questions," "conjectures," and "inquiries" made by the programmers while reading the code.

Teacher and student alike can benefit from reading this paper, which suggests, perhaps better than any other, what a useful model of program comprehension might be. Examples from the data contribute to one's understanding of the assimilation (reading) process on one hand, yet detract from the author's description of his model on the other. Practical implications need to be drawn by the reader. Asking students what Letovsky's results imply (about documentation, for example) should evoke interesting discussion.

The 1987 reprint includes an appendix, "Other Categories of Questions and Conjectures," which illustrates programmer thinking not accounted for by the author's model.

Letovsky86b

Letovsky, Stanley, and Elliot Soloway. "Delocalized Plans and Program Comprehension." *IEEE Software 3*, 3 (May 1986), 41-48.

The authors conclude, based on the same study as [Letovsky86a], that inadequately documented "delocalized plans" are sometimes responsible for misreading of programs on the part of maintainers. They analyze comprehension failures by their subjects and suggest techniques to prevent such misunderstandings when composing programs.

According to this paper, the task of understanding a program is one of uncovering the intention behind the code. Intentions are described as "goals." Techniques for realizing goals in a particular implementation are called "plans." Plans are a lot like algorithms, but they may involve non-contiguous elements and may be combined in ways we do not usually consider for algorithms. Two plans involving loops may be combined into a solution using a single loop implementing two distinct goals, for example.

The authors have observed that readers of programs tend to infer the goals of code fragments on the basis of locally available information. If the plan for a fragment is "delocalized," that is, part of the plan is realized in non-contiguous code, the reader will often incorrectly perform this inference. The authors suggest various documentation techniques to mitigate reading problems resulting from delocalized plans, most of which re-

quire the programmer to be more explicit in comments about his intentions. The paper also includes a brief section on related work and tools for assisting program reading.

The comprehension difficulties discussed here are not surprising ones, yet the paper comes as something of a revelation to most of us who have never thought much about those difficulties or have never thought about them so clearly. This is "must" reading for student and teacher alike.

Levine90

Levine, Linda, Linda H. Pesante, and Susan B. Dunkle. *Technical Writing for Software Engineers.* Curriculum Module SEI-CM-23, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., May 1990.

Capsule Description: *This module, which is directed specifically to software engineers, discusses the writing process in the context of software engineering. Its focus is on the basic problem-solving activities that underlie effective writing, many of which are similar to those underlying software development. The module draws on related work in a number of disciplines, including rhetorical theory, discourse analysis, linguistics, and document design. It suggests techniques for becoming an effective writer and offers criteria for evaluating writing.*

This curriculum module is a brief presentation of what software engineers should know about written technical communication. Levine, Pesante, and Dunkle believe that future software engineers must be taught to write well. They provide substantial advice to instructors who may be sympathetic to this idea but who are uncertain of what they can do to implement it. This is an essential resource for teachers who sincerely want their students to write better.

Linger79

Linger, Richard C., Harlan D. Mills, and Bernard I. Witt. *Structured Programming: Theory and Practice.* Reading, Mass.: Addison-Wesley, 1979.

An extended apology for and explication of structured programming. The book provides a good description and adequate examples of the algorithmic conversion of arbitrary programs into structured ones. Of greatest interest for our purposes is the 66-page Chapter 5, "Reading Structured Programs." The entire book is sprinkled with exercises.

Much of Chapter 5 is devoted to an example of how an unstructured, undocumented program can be structured and documented bottom-up using *stepwise abstraction*, the formulation of an abstract description of what a fragment does from the fragment itself. (The techniques described here provide the basis for what Basili and Mills do, with greater formality, in [Basili82].) The chapter is notable for its insights into program reading generally and into the proper nature of comments. According to the authors, well-documented programs can largely be read top-down, whereas poorly documented programs have to be read mostly bottom-up. In practice, they suggest, both strategies are usually used, even for well-documented or totally mysterious programs.

Teachers should read Chapter 5 and whatever other sections they find of interest. "Reading Structured Programs" is full of practical ideas useful when teaching program reading. Students who do not need to be convinced of the virtues of structured programming and who do not need to structure spaghetti code are likely to find this book tedious.

Littman86

Littman, David C., Jeannie Pinto, Stanley Letovsky, and Elliot Soloway. "Mental Models and Software Maintenance." In *Empirical Studies of Programmers: Papers Presented at the First Workshop on Empirical Studies of Programmers, June 5-6, 1986,*

Washington, D.C., Elliot Soloway and Sitharama Iyengar, eds. Norwood, N.J.: Ablex, 1986, 80-98. Reprinted in *J. Syst. and Software 7*, 4 (Dec. 1987), 341-355.

Abstract: *Understanding how a program is constructed and how it functions are significant components of the task of maintaining or enhancing a computer program. We have analyzed videotaped protocols of experienced programmers as they enhanced a personnel data base program. Our analysis suggests that there are two strategies for program understanding, the* systematic *strategy and the* as-needed *strategy. The programmer using the systematic strategy traces data flow through the program in order to understand global program behavior. The programmer using the as-needed strategy focuses on local program behavior in order to localize study of the program. Our empirical data show that there is a strong relationship between using a systematic approach to acquire knowledge about the program and modifying the program successfully. Programmers who used the systematic approach to study the program constructed successful modifications; programmers who used the as-needed approach failed to construct successful modifications. Programmers who used the systematic strategy gathered* knowledge about the causal interactions of the program's functional components. *Programmers who used the as-needed strategy did not gather such causal knowledge and therefore failed to detect interactions among components of the program.*

This empirical study reports on 10 professional programmers performing the maintenance task described in [Letovsky86a]. (Why the two papers report different numbers of subjects is not explained.) The authors argue that different subjects applied different strategies to the program reading task. The key to successfully modifying the target program was understanding interactions among components—presumably a not uncommon situation—and only subjects who attempted to understand the overall operation of the program were successful. Significantly, years of experience was correlated neither with successful modification nor systematic strategy. The authors admit that the subjects might have behaved differently had they been able to test and debug, and they also admit that it is impractical to try to understand truly large programs completely before attempting modification. Nonetheless, they speculate that effective program reading prior to changing any code leads to more efficient maintenance.

This paper is perhaps most useful as a means of sending a warning to students inclined to begin modifying a program before they understand it.

Lukey81

Lukey, F. J. "Comprehending and Debugging Computer Programs." In *Computer Skills and the User Interface*, M. J. Coombs and J. L. Alty, eds. London: Academic Press, 1981, 201-219.

Lukey reviews and comments on progress in understanding program comprehension and debugging. The author suggests that there are three methods of studying these phenomena—the experimental approach, the artificial intelligence approach, and the observational approach. He focuses on the former two.

This is a good summary of a large body of research, much of which, because of limitations of space, we have not been able to include here. Although, as a literature review, this material is somewhat out of date, it provides helpful coverage of early references. Lukey's review is recommended for teachers and for students with serious interest in studying program comprehension.

Miara83

Miara, J. Richard, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. "Program Indentation and Comprehensibility." *Comm. ACM 26*, 11 (Nov. 1983), 861-867.

Abstract: *The consensus in the programming community is that indentation aids program comprehension, although many studies do not back this up. We tested program comprehension on a Pascal program. Two styles of indentation were used—blocked and nonblocked—in addition to four possible levels of indentation (0, 2, 4, 6 spaces). Both experienced and novice subjects were used. Although blocking style made no difference, the level of indentation had a significant effect on program comprehension. (2-4 spaces had the highest mean score for program comprehension.) We recommend that a moderate level of indentation be used to increase program comprehension and user satisfaction.*

This paper addresses the impact of indentation and blocking on program comprehension. By blocked indentation, the authors mean that statements immediately within a **begin** ... **end** pair share a common left margin with those delimiting keywords. In nonblocked style, the delimited statements are indented further. The authors review previous studies of indentation, noting that their support for the hypothesis that program indentation aids program readability and comprehension is, at best, ambiguous. They explain possible reasons for the discrepancy between their results and those reported by others. The results of the experiments reported here favor the view that indentation aids comprehension, but they also show that excessive indentation (6 or more spaces) does not increase the effect. Interestingly enough, the novices in the study reacted very favorably to indented code and rejected the nonindented program. Experts, on the other hand, showed no such prejudice.

This is a good paper on the effect of formatting practices. The experiment was done carefully, which makes it especially relevant to those interested in empirical studies. The practical wisdom to take away from the paper is simple: indent code three spaces to show its structure. This paper is recommended for both instructors and students.

Moffat84

Moffat, David V. *Common Algorithms in Pascal with Programs for Reading.* Englewood Cliffs, N.J.: Prentice-Hall, 1984.

A discussion of basic algorithms illustrated with Pascal examples. The book is distinguished by the inclusion of complete, documented programs to accomplish simple tasks. It makes pleasant bedtime reading for any programmer interested in clear, straightforward coding and documentation. Reading exercises are included with the complete programs, and teachers may find these useful. Many questions are too broad and open-ended for students not accustomed to program reading, however.

Oman90a

Oman, Paul W., and Curtis R. Cook. "The Book Paradigm for Improved Maintenance." *IEEE Software 7*, 1 (Jan. 1990), 39-45.

A condensed version of [Oman90b], but with a few useful ideas and comments not found in that more scholarly paper. Its brevity makes it attractive for student reading.

Oman90b

Oman, Paul W., and Curtis R. Cook. "Typographic Style is More than Cosmetic." *Comm. ACM 33*, 5 (May 1990), 506-520.

The authors discuss relatively straightforward source code formatting and commenting techniques to improve program comprehension. They also discuss their experimental evidence to support their claim that the techniques do, in fact, achieve their objective.

Following a brief review of the literature, Oman and Cook introduce their "book format paradigm" as a vehicle for displaying source code. They point out that the organization of books into chapters, sections, and paragraphs, supplemented by prefaces, tables of con-

tents, and indexes, is both familiar and serviceable. Similar organization and devices can be applied to computer programs, and most of the application can be done automatically.

A program formatted according to the book paradigm includes a preface, table of contents, chapter divisions, pagination, and indices. Most of the added text is in the form of comments, although page headings, which include chapter name and page number, are apparently generated by a listing program. In the table of contents, for example, one might find the main procedure listed as "Chapter 2," beginning on page 4. A procedure called on page 4 would be listed in a module index at the end of the program as being called from the main procedure; and, if it called other modules, those would also be noted.

The authors refer to features that aid the reader in finding his way around the program as "macro-typographic" factors. They also discuss "micro-typographic" factors, including the addition of blank lines and indentation, vertical alignment conventions, use of upper- and lowercase, use of boldface, addition of "paragraphing" (putting more than one statement on a line), etc. All these conventions were selected by appeal to "typographic style principles," which the authors claim is supported by the empirical evidence.

The paper also reports on four experiments carried out with Pascal and C code formatted according to the book paradigm or formatted conventionally. Subjects were asked to implement an enhancement, complete a comprehension test, or complete a call graph for the code in question. Data were also gathered through think-aloud protocols in one of the experiments. In each case, subjects using book format code outperformed their fellow subjects using more conventional program listings. Moreover, the authors report that they adapted easily to the book format, even in the absence of instruction in its use.

Oman and Cook conclude that their book paradigm is natural and useful, though they recommend that students be taught style principles rather than particular formatting conventions.

This is an important and thought-provoking paper. It is easy to quibble about the specifics of the book format paradigm, but it is difficult to dismiss the thrust of this work. The authors' recommended conventions are by no means radical, and require no substantive changes in executable statements of the code. Yet, the claimed results are impressive. This is "must" reading for teachers. Students not specifically interested in empirical studies should probably read [Oman90a] instead.

Pazel89

Pazel, D. P. "DS-Viewer—An Interactive Graphical Data Structure Presentation Facility." *IBM Systems J. 28*, 2 (1989), 307-323.

Abstract: *DS-Viewer is a tool that is the result of a research project in data structure presentation within a program state. This tool addresses two distinct issues in this area: (1) to effectively present data structures themselves for a given program state and (2) to present groups of data structures and their interrelationships as described by their pointer definitions. Graphical presentations were developed to address these issues. For the data structure presentation, the user is provided a display window for any single data structure instance formatted with its fields and field values. Flexibility in display is provided by allowing the user a choice from the various value formats for each field. For groups of data structure instances, a graphical drawing space is provided in which pictures of these data structure instances and their interrelationships are drawn as blocks and arrows. The computer assists the user in drawing such a picture by describing its components, allowing the user to choose which to draw and to construct as much of the picture as desired.*

This paper describes an IBM prototype system that runs on a PC as a Microsoft Windows application. The tool is designed for use in debugging complex data structures that may themselves have been corrupted. DS-Viewer allows the user to select interactively the data structure components and representations to be used for the graphic presentation. It

allows the user to select a display of multiple instances of data structures linked by pointers.

Pennington87

Pennington, Nancy. "Comprehension Strategies in Programming." In *Empirical Studies of Programmers: Second Workshop*, Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, eds. Norwood, N.J.: Ablex, 1987, 100-113.

Abstract: *This report focuses on differences in comprehension strategies between programmers who attain high and low levels of program comprehension. Comprehension data and program summaries are presented for 40 professional programmers who studied and modified a moderate length program. Illustrations from detailed think-loud protocol analyses are presented for selected subjects who displayed distinctive comprehension strategies. The results show that programmers attaining high levels of comprehension tend to think about* both *the program world and the domain world to which the program applies while studying the program. We call this a* cross-referencing *strategy and contrast it with strategies in which programmers focus on program objects and events* or *on domain objects and events, but not both.*

Based on her previous research, Pennington proposes that understanding of overall program flow control precedes the more detailed understanding of program functions. In particular, she suggests that program readers build at least two mental models of the program they are studying, a "program model" and a "domain model." The program model is characterized by an abstract knowledge of the program's text structures. The domain model relates objects and functions in the problem domain to source-language entities.

The author carried out an experiment using a minimally documented 200-line FORTRAN program. Subjects were asked to study the program for 45 minutes in preparation for a modification task. Some of the subjects were asked to "think aloud" as they examined the program. After the study period, subjects wrote summaries explaining what the program did and answered 20 comprehension questions. They were given an additional 30 minutes to implement the requested change, after which a second summary was written and 20 more comprehension questions answered. Using her analysis of the data, Pennington asserts that the comprehension strategies of the subjects can be characterized as "program-level," "domain," or "cross-referencing," the latter being a strategy that combines features of the other two. That is, the programmers concentrated either on the program, on the problem domain, or somehow effectively related the two. Not surprisingly, it was the cross-referencing readers who performed best.

Whether or not Pennington's results indicate that program readers create two distinct mental models in succession, they certainly support the layered abstractions proposed by Brooks [Brooks83] and Letovsky [Letovsky86a]. This is an insightful paper discussing the cognitive process of program comprehension. It is equally interesting on methodological grounds. Pennington's paper is recommended reading for instructors interested in program comprehension. Student benefits from reading the paper, however, may be limited.

Shneiderman79

Shneiderman, Ben, and Richard Mayer. "Syntactic Semantic Interactions in Programmer Behavior: A Model and Experimental Results." *Intl. J. Comp. & Info. Sciences 8*, 3 (June 1979), 219-238.

Abstract: *This paper presents a cognitive framework for describing behaviors involved in program composition, comprehension, debugging, modification, and the acquisition of new programming concepts, skills, and knowledge. An information processing model is presented which includes a long-term store of semantic and syntactic knowledge, and a working memory in which problem solutions are constructed. New experimental evidence is presented to support the model of syntactic/semantic interaction.*

The authors present their cognitive model of programmer behavior, the "syntactic/semantic" model. They suggest that this model is useful in explaining a variety of behaviors, including program reading and program writing. The authors hypothesize that programmers retain both semantic and syntactic knowledge in long-term memory, and that they use short-term and working memories in performance of various program-related tasks. Semantic knowledge and syntactic knowledge are largely independent in this model. Semantic knowledge is multilayered and substantially language-independent; syntactic knowledge applies to particular programming languages. Shneiderman and Mayer describe how their model applies to program reading, program writing, debugging, and learning programming languages. They conclude their paper with brief discussions of experiments that they offer as supporting evidence for their theory.

In program comprehension, according to this theory, the reader "constructs a multileveled internal semantic structure to represent the program," a process of encoding from the program syntax, which is not memorized directly. The internal structure is built by recognizing the function of program components and fragments as "chunks." These pieces are then aggregated until a description of the entire program is available.

This is a paper everyone should read. It presents a typical cognitive model in an approachable way, and shows how such models are used and verified. It also offers insight into programmer behavior. Yet, the structural complexity of the syntactic/semantic model makes the model seem less useful than it should be, primarily because a totally adequate model would be very much richer in processing details. The processes reified in this model are largely implicit in other comprehension models. Shneiderman's and Mayer's mental model of a program is quite similar to that of Brooks [Brooks78] and Letovsky [Letovsky86a]. Their description of the assimilation process, however, is strictly bottom-up.

Shneiderman80

Shneiderman, Ben. *Software Psychology: Human Factors in Computer and Information Systems*. Cambridge, Mass.: Winthrop, 1980.

Software Psychology is a handbook for the application of psychology to computer-related issues. Shneiderman provides a crash course on methods of psychological research and proceeds to discuss topics from program reading to team organization and the design of interactive systems. Although this volume was written a decade ago, it remains an invaluable reference on psychological factors related to the computer. The book contains an extensive bibliography.

Tenny88

Tenny, Ted. "Program Readability: Procedures Versus Comments." *IEEE Software 14*, 9 (Sept. 1988), 1271-1279.

Abstract: *A 3 × 2 factorial experiment was performed to compare the effects of procedure format (none, internal, or external) with those of comments (absent or present) on the readability of a PL/I program. The readability of six editions of the program, each having a different combination of these factors, was inferred from the accuracy with which students could answer questions about the program after reading it. Both extremes in readability occurred in the program editions having no procedures: without comments the procedureless program was the least readable and with comments it was the most readable.*

An interesting paper that defines readability within the context of maintenance: "a program is readable if information needed to maintain it is easily found by reading the code." The author formalizes this definition by expressing readability as the average number of right answers to a series of questions about the program in a given length of time.

The experiment reports that six versions of the same program were used to explore the

effects of comments versus the inclusion of procedures. Four editions of the program included procedures that performed the major subtasks. Both internal and external (*i.e.*, separately compiled) procedure definitions were used. Two of the programs were procedureless. Commented and uncommented versions of each program version were used as well. The same set of questions accompanied each of the programs. Scores were tabulated, and ANOVA and *F*-tests were performed to determine the statistical significance of the differences between the mean scores.

The reported results are somewhat surprising. The procedureless program with comments was the least readable, whereas the same program with no comments was the most readable. As far as this particular program is concerned, however, the author concludes that procedures have little effect on readability, whereas comments do seem to have an effect. Yet, there are compelling reasons to believe that a large program is more readable with the modules expressed as separate procedures. Thus, "[While] it would be unwise to extrapolate these results to all programs, they do indicate that procedures can have little effect on the readability of programs below a certain size." The results reported by the author differ qualitatively from results obtained by himself on a previous experiment in which the procedureless program got higher scores than the program with internal procedures, with or without comments. Possible explanations for these differences are explored.

Aside from the statistical value of this experiment, the author's questions (which are included in the paper) are of much pedagogical value. Instructors are encouraged to read it. This information may be of limited value to beginning students. Advanced students may find this paper interesting nevertheless.

Thomas90

Thomas, E. J., and Paul W. Oman. "A Bibliography of Programming Style." *ACM SIGPLAN Notices 25*, 2 (Feb. 1990), 7-16.

A lightly annotated bibliography of nearly 100 references on programming style, broadly construed. The Thomas and Oman serves as a helpful complement to this bibliography.

Weinberg71

Weinberg, Gerald M. *The Psychology of Computer Programming.* New York: Van Nostrand Reinhold, 1971.

Weinberg devotes the first chapter of his well-known book to program reading, remarking ruefully that "[e]ven programmers do not read programs." He suggests that there is much to learn from reading both good and bad programs. Most of the chapter is devoted to examples of the factors affecting what actually gets coded: limitations of the machine, the implementation language, and the programmer; historical accidents; and evolving specifications.

Weiser81

Weiser, Mark. "Program Slicing." *Proc. 5th Int. Conf. on Software Eng.* New York: IEEE, 1981, 439-449.

Abstract: *Program slicing is a method used by experienced computer programmers for abstracting from programs. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a "slice", is an independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behavior.*

Finding a slice is in general unsolvable. A dataflow algorithm is presented for approximating slices when the behavior subset is specified as the values of a set of variables at a statement. Experimental evidence is presented that these slices are used by programmers

during debugging. Experience with two automatic slicing tools is summarized. New measures of program complexity are suggested based on the organization of a program's slices.

Being able to find a program slice simplifies analysis of a program. Even though program slicing cannot be fully automated, the concept of a slice is a useful one.

Weiser explains slicing by pointing out that, when fixing a bug, an experienced programmer usually focuses only on those parts of the program that may obviously have something to do with the bug in question. Other parts of the program are ignored, effectively having been deleted in the programmer's mind from the code being studied. Programmers apply this same technique when making program improvements or modifications.

The paper considers the slicing of block-structured programs written in a Pascal-like language. A slice must have two desirable properties: (1) it must have been obtained from the original program by statement deletion, and (2) the behavior of the slice must be the same as that of the original program, as observed through the domain of the specified subset of behavior. Characterizations of programs in terms of flow graphs are explained, and meaning is given to a slice within those contexts. To make the problem of finding a program's slice tractable, Weiser introduces a weaker definition of slice and gives sufficient conditions for statement inclusion. Weiser also introduces a number of slice-based complexity metrics and discusses their computation.

The paper is quite technical and is recommended only for teachers and advanced students. It does, however, provide a name for and some analysis of an intuitive, widely used comprehension strategy.

Wilde89

Wilde, Norman, and Stephen M. Thebaut. "The Maintenance Assistant: Work in Progress." *J. Syst. and Software 9*, 1 (Jan. 1989), 3-17.

Abstract: *The Maintenance Assistant project at the Florida/Purdue Software Engineering Research Center seeks to develop methodologies and tools in the complex tasks associated with making changes to software systems. Three broad approaches are currently being explored:* dependency analysis *involves capturing the dependencies between different entities in a software system and the development of tools to present and analyze these dependencies.* Reverse engineering *involves the identification or "recovery" of program requirements and/or design specifications that can aid in understanding and modifying it.* Program change analysis *involves methods for analyzing differences between two versions of a program in order to understand a change that has been made and detect possible maintenance-induced errors. A strength of the project has been the very close relationship with the industrial affiliates of the Software Engineering Research Center. It is hoped that these organizations will be able to apply the methodologies currently being explored in their own software projects and in tools to be used by their clients.*

This paper surveys a number of program maintenance techniques currently in use in industry and under prototype development at the Software Engineering Research Center (SERC). The work described is expected to produce tools that are language-independent, semi-automatic (with human interaction required), and potentially applicable to programs of any size.

The author discusses four broad classifications of dependency analysis: data flow dependencies, definition dependencies, calling dependencies, and functional dependencies. A prototype tool is under development to assist the programmer in exploring these dependencies. Components of the system all utilize a single program database. The prototype handles only C programs.

SERC's reverse engineering effort focuses on identifying a useful model of program comprehension. The initial goal is to establish a framework for identifying and assessing the

effectiveness of strategies and techniques that either aid the comprehension process directly or partially automate it. In connection with this, SERC has surveyed some 120 program reading tools currently in use. A summary of their findings is presented in the paper.

Finally, program change analysis to assess the impact of program change is discussed. Change analysis tools can be used to help programmers identify unexpected side effects, to guide management in the allocation of resources, or to gauge the system's vulnerability to newly introduced errors. The paper reports on SERC's strategies based on incremental data flow analysis.

This paper does a good job of surveying existing tools and suggesting the nature of those that might become available in the future. Recommended reading for teachers and advanced students.

Wilde90

Wilde, Norman. *Understanding Program Dependencies.* Curriculum Module SEI-CM-26, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Aug. 1990.

Capsule Description: *A key to program understanding is unraveling the interrelationships of program components. This module discusses the different methods and tools that aid a programmer in answering the questions: "How does this system fit together?" and "If I change this component, what other components might be affected?"*

This curriculum module discusses some of the important relationships that may exist among elements of a program. Wilde briefly discusses program comprehension and what is known about it. The bulk of the module treats dependencies among data items, types, program units, and source files: what they are, how to find them, how they can be presented to the program reader, and what tools are available to help the reader deal with them.

The author's interest is principally in what the maintainer needs to know about how program components work together. Even within this context, Wilde's scope is narrow. Nonetheless, this module is useful in its making explicit some of what the program reader may need to learn from a program.

Like all SEI curriculum modules, this report is addressed to teachers, although its concise overview may appeal to students as well.

Acknowledgements

A number of people contributed in small or large ways to the production of this report. Gary Ford's commitment to producing educational materials led us to consider a report on program reading. Special thanks are due Mark Ardis, who said "do it" at a time when a more ambitious project encountered administrative snags, and who, along with Nancy Mead, brought the authors together.

Other members of the SEI staff who gave of their time to talk about program reading, teaching, Ada, and program documentation are: Edward Averill, Judy Bamberger, Len Bass, Mike Christel, Linda Levine, Jim Tomayko, Rob Veltre, Nelson Weiderman, and Greg Zelesnik. Similar thanks are due Jon Bentley and Chris Van Wyk of AT&T Bell Laboratories; David Bustard, SEI visiting scientist from the University of Ulster; Curtis Cook, of Oregon State University; John Cross, of Indiana University of Pennsylvania; David Dobkin, of Princeton University; Rick Green, of Carnegie Mellon University; Phil Kulton, of Harris Corporation; Paul Oman, of the University of Idaho; Ruth Shapiro, SEI resident affiliate from GTE Government Systems Corporation; and Norman Wilde, SEI visiting scientist from the University of West Florida.

Chuck Engle, formerly an SEI resident affiliate from the U.S. Army and now of Florida Institute of Technology, was a tireless Ada oracle and program critic. Our effort could not have been successful without SEI librarians Karola Fuchs and Sheila Rosenthal. Linda Pesante did her usual splendid job of editing, which seems even more impressive from the vantage point of SEI author than from that of interested bystander. She was assisted by Marie Elm, who was tenacious in her pursuit of clarity. Angela Wilkerson typed in some of the abstracts and edits.

We have no doubt missed a few names, for which we apologize.

Our production schedule was short, so virtually all of the above people can credibly deny responsibility for the final words we have put on paper. We thank them all and accept full responsibility for any mistakes.

Appendix: Program Source Code

On the following pages can be found the Ada source code for the program on which the reading exercises are based. The code is shown in alphabetical order by file name. The file names are those from the UNIX environment in which the program was developed. Package specifications and bodies are separated. The file naming convention is as follows: The main procedure and all package specifications are in file *name.a*, where the procedure or package is *name*. Package and task bodies are in file *name_body.a*, where the package or task is *name*.

Contents of the files appear on the pages shown below:

```
--============================================================================
--|
--| TASK NAME: numbers.search.(do_search_type)
--|
--| ALGORITHM/STRATEGY: Wait for call to either entry enter_start_search or
--|    entry enter_start_search_from_checkpoint.  In the former case, a call
--|    is made to procedure start_search to begin the search, after which the
--|    wait is resumed.  If enter_start_search_from_checkpoint is called,
--|    the checkpoint file is read, and procedure start_search is called with
--|    restarting=true.
--|
--| NOTES: Task normally terminates using a terminate option in select
--|    statement.
--|
--============================================================================

WITH other_io;
WITH sequential_io;
WITH text_io;
WITH timekeeper;

SEPARATE (numbers.search)

TASK BODY do_search_type IS

   -- Range of count of digit positions in number
   SUBTYPE number_of_occurrences_range IS natural
      RANGE 0 .. maximum_number_length;

   -- To hold counts of number of occurrences of each digit for checkpoint
   TYPE wide_select_vector_type        IS ARRAY(radix_digit) OF
      number_of_occurrences_range;

   -- Definition of type to record checkpoint
   TYPE checkpoint_record              IS
      RECORD

         -- Order of checkpointed search
         order               : natural;

         -- Length of PDI sought in checkpointed search
         search_length       : digit_range;

         -- Value of smin at checkpoint (see start_search)
         smin                : number;

         -- Number of digits not yet determined at checkpoint
         free                : number_of_occurrences_range;

         -- Distribution of digits already selected at checkpoint
         select_vector       : wide_select_vector_type;

         -- Next digit to be added to distribution
         digit_to_place      : radix_digit;

         -- Elapsed time of search prior to checkpoint
         elapsed_time        : duration;

         -- Number of combinations tested prior to checkpoint
         combinations_tested : natural;

      END RECORD;
```

```
-- Package to perform I/O of checkpoint records
PACKAGE checkpoint_io IS NEW sequential_io (element_type =>
   checkpoint_record);

-- Checkpoint record
checkpoint               : checkpoint_record;
-- Checkpoint file
checkpoint_file          : checkpoint_io.file_type;
-- One second
second                   : CONSTANT duration := 1.0;
-- Time between checkpoints
checkpoint_interval      : duration := 3600*second;
-- Flag to tell start_search if search is being restarted from a
-- checkpoint (true if it is)
restarting               : Boolean := false;

-- Uninitialized time value imported from package timekeeper
uninitialized_time       : CONSTANT timekeeper.time :=
                             timekeeper.uninitialized_time;
-- Search start time or time of last checkpoint
t0                       : timekeeper.time := uninitialized_time;
-- Current time
t1                       : timekeeper.time;

-- Exception raised for any error related to opening input file
file_open_error          : EXCEPTION;
```

```
--============================================================================
--|
--| PROCEDURE NAME: numbers.search.(do_search_type).start_search
--|
--| PURPOSE: To perform PDI search.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 6/11/90
--| DATE OF LAST REVISION: 8/9/90          VERSION: 1.2
--|
--| PARAMETERS:
--|    radix                      (in) radix of PDIs sought
--|
--|    order                      (in) order of PDIs sought
--|
--|    search_length              (in) length of PDIs sought
--|
--| INPUT: None.
--|
--| OUTPUT: (To default file) Information about the search before
--|    and after it is completed.  Procedure select_digits prints out
--|    PDIs found and other intermediate information.
--|
--| ASSUMPTIONS/LIMITATIONS: None.  Search may fail due to inadequate
--|    storage allocation.
--|
--| ALGORITHM/STRATEGY: The real search is carried on by select_digits.
--|    This procedure merely initializes the search and outputs a summary
--|    at the end.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: In a length-search_length search, any combination of
--|    search_length base-radix digits is a potential PDI.  To
--|    determine if it is actually a PDI, all we need do is raise each
--|    digit to the order power and sum the result.  If the sum
--|    has the same combination of digits we began with, then we have found
--|    a PDI.  Procedure select_digits builds up a candidate combination
--|    of digits by selecting the number of radix-1 digits in
--|    the combination, then the number of radix-2 digits, etc.  The
--|    array select_vector keeps track of how many of each digit have been
--|    put in the trial combination (select_vector(i) is the number of
--|    occurrences of digit i).  Variable smin holds the minimum possible
--|    value of any PDI found using the current combination of digits,
--|    that is, the sum of the order powers of all the digits so far
--|    placed in the trial digit combination.  Table look-up is used
--|    in the computation of smin.  Array sum_table(i, j) contains the
--|    the sum of i instances of j raised to the order power.
--|
--============================================================================

PROCEDURE start_search (radix : IN radix_range; order : IN natural;
   search_length : IN digit_range) IS
```

```
        -- Number of complete digit combinations tested in PDI search
        combinations_tested    : natural;

        -- Default and current number of calls to procedure select_digits
        -- between checks of whether the user has requested service from the
        -- keyboard (see select_digits)
        default_max_calls_to_select_digits
                               : CONSTANT natural := 3000;
        current_max_calls_to_select_digits
                               : natural := default_max_calls_to_select_digits;

        -- Number of calls to select_digits since check was made for keyboard
        -- interrupt
        calls_to_select_digits : natural := 0;

        -- Valid digits in base-radix
        SUBTYPE numeral              IS radix_digit RANGE 0 .. radix - 1;

        -- Range of number of occurrences of any digit possible for this search
        SUBTYPE number_of_occurrences IS natural RANGE 0 .. search_length;

        -- To hold information on digit combinations
        TYPE select_vector_type      IS ARRAY(numeral) OF number_of_occurrences;

        -- Characters used to set off related information in output file
        divider                : CONSTANT string := "--------------------";

        -- Current combination of digits selected for consideration
        select_vector          : select_vector_type := (numeral => 0);

        -- Minimum value of PDI possible with combination of digits derived
        -- from current combination specified by select_vector
        smin                   : number;

        -- Table of sums of powers of base-radix digits
        sum_table              : ARRAY(number_of_occurrences, numeral) OF number;

        -- Current time obtained from package timekeeper
        t                      : timekeeper.time;

        -- Import operators from package timekeeper
        FUNCTION "-" (x : timekeeper.time; y : timekeeper.time) RETURN duration
           RENAMES timekeeper."-";
        FUNCTION "=" (x : timekeeper.time; y : timekeeper.time) RETURN Boolean
           RENAMES timekeeper."=";
```

```
--============================================================================
--|
--| PROCEDURE NAME:
--|    numbers.search.(do_search_type).start_search.initialize_table
--|
--| PURPOSE: To initialize table of sums of powers of base-radix digits.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 6/11/90
--| DATE OF LAST REVISION: 8/8/90          VERSION: 1.1
--|
--| PARAMETERS:
--|    radix                      (in) radix of PDIs sought
--|
--|    order                      (in) order of PDIs sought
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: It is assumed that initialization of
--|    sum_table proceeds without overflow.
--|
--| ALGORITHM/STRATEGY: The initialization is straightforward, using
--|    the operations on multiple-precision integers defined in package
--|    numbers.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: See description of sum_table in start_search.
--|
--============================================================================

PROCEDURE initialize_table (radix : IN radix_range;
   order : IN natural) IS
```

```
        BEGIN -- initialize_table

           -- Initialize sums (all 0) for any number of 0 digits
           FOR count IN number_of_occurrences LOOP
              sum_table(count, 0) := make_number(radix, 0);
              END LOOP;

           -- Initialize sums (all 0) for 0 instances of any other digit
           FOR digit IN 1 .. radix - 1 LOOP
              sum_table(0, digit) := make_number(radix, 0);
              END LOOP;

           -- Initialize sum (1) for 1 instance of 1 digit
           sum_table(1, 1) := make_number(radix, 1);

           -- Initialize entries of row 1 of sum_table that have not yet been
           -- given values
           FOR digit IN 2 .. radix - 1 LOOP
              sum_table(1, digit) := make_number(radix, digit);
              FOR power IN 2 .. order LOOP
                 sum_table(1, digit) := digit * sum_table(1, digit);
              END LOOP;
           END LOOP;

           -- Initialize entries in remaining rows of sum_table that have not
           -- yet been given values
           FOR count IN 2 .. search_length LOOP
              FOR digit IN 1 .. radix - 1 LOOP
                 sum_table(count, digit) := sum_table(count-1, digit) +
                    sum_table(1, digit);
              END LOOP;
           END LOOP;

        END initialize_table;
```

```
--=========================================================================
--|
--| FUNCTION NAME: numbers.search.(do_search_type).start_search.is_pdi
--|
--| PURPOSE: To determine if a number is a PDI.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 6/11/90
--| DATE OF LAST REVISION: 8/8/90          VERSION: 1.1
--|
--| PARAMETERS:
--|    smin                       (in) sum of powers of digits
--|                               specified by select_vector
--|                               (candidate PDI)
--|
--|    select_vector              (in) combination of digits in smin
--|
--| RETURNS: True if smin is a PDI, false otherwise.
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ALGORITHM/STRATEGY: Check if the combination of digits in smin
--|    is the same as in select_vector.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: None.
--|
--=========================================================================

FUNCTION is_pdi (smin : IN number; select_vector : IN select_vector_type)
   RETURN Boolean IS

   -- Base-smin.radix digit from smin
   digit : numeral;

   -- Variable in which digit combination of smin is computed
   tally : select_vector_type := (numeral => 0);

BEGIN -- is_pdi

   -- Compute digit combination in smin
   FOR position IN 1 .. smin.high_order_digit LOOP
      digit := smin.digit(position);
      tally(digit) := tally(digit) + 1;
   END LOOP;

   -- Return true smin has same digit combination as that specified by
   -- select_vector
   RETURN (tally = select_vector);

END is_pdi;
```

```
--=========================================================================
--|
--| PROCEDURE NAME:
--|    numbers.search.(do_search_type).start_search.report_pdi
--|
--| PURPOSE: To announce a number as a PDI.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 6/11/90
--| DATE OF LAST REVISION: 8/8/90          VERSION: 1.2
--|
--| PARAMETERS:
--|    pdi                         (in) PDI to be output
--|
--| INPUT: None.
--|
--| OUTPUT: (To default file) number, identification, elapsed time,
--|    and number of combinations tested.
--|
--| ASSUMPTIONS/LIMITATIONS: No provision is made for too short an
--|    output line.
--|
--| ALGORITHM/STRATEGY: Function numbers.convert_to_string is used to
--|    obtain a representation of pdi to be output.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: Identifies number as PDI or PPDI.
--|
--=========================================================================

PROCEDURE report_pdi (pdi : IN number) IS
```

```
   -- Character representation of pdi
   pdi_string : dynamic_string;

BEGIN -- report_pdi

   -- Obtain and output character representation of pdi
   pdi_string := convert_to_string(pdi);
   text_io.put (pdi_string.char(1 .. pdi_string.length));
   text_io.new_line;

   -- Output its identification
   text_io.put ("is an order-");
   other_io.natural_io.put (order, width => 1);
   IF (order=search_length) THEN
      text_io.put (" PPDI.");
   ELSE
      text_io.put (", length-");
      other_io.natural_io.put (length_of_number(pdi), width => 1);
      text_io.put (" PDI.");
   END IF;
   text_io.new_line;

   -- Output elapsed time of search
   text_io.put ("Found after: ");
   timekeeper.elapsed_time;
   text_io.new_line;

   -- Output number of combinations of digits tested
   text_io.put ("Number of combinations tested: ");
   other_io.natural_io.put (combinations_tested, width => 1);
   text_io.new_line(2);

END report_pdi;
```

```
--=========================================================================
--|
--| PROCEDURE NAME:
--|    numbers.search.(do_search_type).start_search.print_state
--|
--| PURPOSE: To display the current state of the PDI search.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 6/11/90
--| DATE OF LAST REVISION: 8/9/90          VERSION: 1.2
--|
--| PARAMETERS:
--|    smin                       (in) current value of smin (see
--|                               start_search)
--|
--|    digit_to_place             (in) next digit to be selected for
--|                               trial digit combination
--|
--|    select_vector              (in) current digits in trial
--|                               combination
--|
--| INPUT: None.
--|
--| OUTPUT: (To default file) number, identification, elapsed time,
--|    and number of combinations tested.
--|
--| ASSUMPTIONS/LIMITATIONS: No provision is made for too short an
--|    output line.
--|
--| ALGORITHM/STRATEGY: Function numbers.convert_to_string is used to
--|    obtain a representation of smin to be output.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: None.
--|
--=========================================================================

PROCEDURE print_state (smin : IN number; digit_to_place : IN numeral;
   select_vector : IN select_vector_type) IS
```

```
   -- Character representation of smin
   smin_string : dynamic_string;

BEGIN -- print_state

   -- Obtain and output character representation of smin
   smin_string := convert_to_string(smin);
   text_io.put_line ("smin:");
   text_io.put (smin_string.char(1 .. smin_string.length));
   text_io.new_line;

   -- Output next digit to be selected in digit combination
   text_io.put ("digit_to_place: ");
   other_io.natural_io.put (digit_to_place, width => 1);
   text_io.new_line;

   -- Output current digit combination
   text_io.put_line ("select_vector:");
   FOR position IN REVERSE numeral LOOP
      text_io.put ("(");
      other_io.natural_io.put (position,
         width => maximum_radix_length);
      text_io.put (") = ");
      other_io.natural_io.put (select_vector(position),
         width => 2);
      text_io.new_line;
   END LOOP;

END print_state;
```

```
--=====================================================================
--|
--| PROCEDURE NAME:
--|    numbers.search.(do_search_type).start_search.record_checkpoint
--|
--| PURPOSE: Output checkpoint information to default output file and
--|    write checkpoint record to checkpoint file.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 6/11/90
--| DATE OF LAST REVISION: 8/8/90          VERSION: 1.1
--|
--| PARAMETERS:
--|    order                       (in) order of PDIs sought
--|
--|    search_length               (in) length of PDIs sought
--|
--|    smin                        (in) current value of smin (see
--|                                start_search)
--|
--|    free                        (in) number of digits yet to be
--|                                selected
--|
--|    select_vector               (in) current digits in trial
--|                                combination
--|
--|    digit_to_place              (in) next digit to be selected for
--|                                trial digit combination
--|
--| INPUT: None.
--|
--| OUTPUT: (To default file) checkpoint information (time, elapsed
--|    time, number of combinations tested, smin, digit_to_place,
--|    and select_vector) and, if necessary, an error message to
--|    the effect that the checkpoint file cannot be written.
--|
--|    (To file "pdi.ckp") checkpoint record.
--|
--| ASSUMPTIONS/LIMITATIONS: No provision is made for too short an
--|    output line.
--|
--| ALGORITHM/STRATEGY: Procedure print_state is used to output
--|    smin, digit_to_place, and select_vector.
--|
--| ERROR CHECKS/RESPONSES: If the checkpoint file cannot be written
--|    for any reason, an error message is output to the default file
--|    and the procedure continues.
--|
--| NOTES: None.
--|
--=====================================================================

PROCEDURE record_checkpoint (order : IN natural; search_length : IN
   digit_range; smin : IN number; free : IN number_of_occurrences;
   select_vector : IN select_vector_type; digit_to_place : IN numeral) IS

   -- Checkpoint record to be written
   checkpoint         : checkpoint_record;

   -- Current time
   checkpoint_time    : timekeeper.time;
```

```
   -- Current digits in trial combination in format for checkpoint
   -- record
   wide_select_vector : wide_select_vector_type := (radix_digit => 0);

BEGIN -- record_checkpoint

   -- Output header and divider
   text_io.put_line ("CHECKPOINT");
   text_io.put_line (divider);

   -- Output time and elapsed time
   timekeeper.time_stamp;
   text_io.new_line;
   text_io.put ("Elapsed time: ");
   timekeeper.get_time (checkpoint_time);
   timekeeper.elapsed_time (checkpoint_time);
   text_io.new_line;

   -- Output number of combinations tested
   text_io.put ("Number of combinations tested: ");
   other_io.natural_io.put (combinations_tested, width => 1);
   text_io.new_line;

   -- Output smin, digit_to_place, and select_vector
   print_state (smin, digit_to_place, select_vector);

   -- Change representation of digit combination for writing checkpoint
   FOR digit IN numeral LOOP
      wide_select_vector(digit) := select_vector(digit);
   END LOOP;

   BEGIN

      -- Create and write checkpoint record
      checkpoint := (order, search_length, smin, free,
         wide_select_vector, digit_to_place, checkpoint_time - t,
         combinations_tested);
      checkpoint_io.create (file => checkpoint_file, mode =>
         checkpoint_io.out_file, name => "pdi.ckp");
      checkpoint_io.write (file => checkpoint_file, item =>
         checkpoint);
      checkpoint_io.close (file => checkpoint_file);

   EXCEPTION

      WHEN OTHERS =>
         text_io.new_line;
         text_io.put_line ("Cannot write to checkpoint file; " &
            "checkpoint ignored");
         text_io.new_line;

   END;

   -- Output divider
   text_io.put_line (divider);
   text_io.new_line;

END record_checkpoint;
```

```
--=====================================================================
--|
--| PROCEDURE NAME:
--|    numbers.search.(do_search_type).start_search.select_digits
--|
--| PURPOSE: To construct combinations of digits, test them as possible
--|    PDIs, and report PDIs found.  Also interacts with task synchronize
--|    to fulfill user requests for information and changes to operating
--|    parameters.
--|
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 6/11/90
--| DATE OF LAST REVISION: 8/9/90          VERSION: 3.1
--|
--| PARAMETERS:
--|    smin                        (in) current value of smin (see
--|                                start_search)
--|
--|    free                        (in) number of digits yet to be
--|                                selected
--|
--|    select_vector               (in) current digits in trial
--|                                combination
--|
--|    digit_to_place              (in) next digit to be selected for
--|                                trial digit combination
--|
--| INPUT: None.
--|
--| OUTPUT: (To default file) search status information, including
--|    checkpoint information and PDIs found.
--|
--|    (To file "pdi.ckp") checkpoint record (through call to
--|    record checkpoint).
--|
--| ASSUMPTIONS/LIMITATIONS: No provision is made for too short an
--|    output line.  On entry, it is assumed that select_vector
--|    represents a combination of digits in the range {radix-1 ...
--|    digit_to_place-1}.  The sum of these digits, each raised to
--|    the appropriate order power, is assumed to be in smin.  On entry,
--|    it is assumed that free digits remain to be selected.
--|
--| ALGORITHM/STRATEGY: Select_digits finds all PDIs that can be
--|    constructed from the combination of digits in select_vector
--|    and digits in the range {digit_to_place ... 0}.  It does so by
--|    calling itself recursively.  Tests are made to prune the search
--|    tree by not considering digit combinations whose corresponding
--|    smin will have too many or too few digits.  Through judicious
--|    use of Boolean restarting, select_digits can reconstruct its
--|    parameters at the time of a checkpoint, as well as its call
--|    stack.  (It does this when it is called with restarting=true.)
--|    At appropriate intervals, checkpoint records are written
--|    through calls to record_checkpoint.  Also at appropriate
--|    intervals, calls are made to synchronize.check_interrupts in
--|    order to process user requests while searches are in progress.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: None.
--|
--=====================================================================
```

```
PROCEDURE select_digits (smin : IN number; free : IN
   number_of_occurrences;  select_vector : IN select_vector_type;
   digit_to_place : IN numeral) IS

   -- Flag to return status from synchronize.check_interrupts
   flag             : interrupt_flag_type;
   -- New checkpoint interval from synchronize.check_interrupts
   interval         : duration;
   -- New maximum number of calls between checks for user interaction
   -- from synchronize.check_interrupts
   max_calls        : natural;

   -- Updated smin to be used in call to select_digits
   new_smin         : number;
   -- Updated select_vector to be used in calls to select_digits
   new_select_vector : select_vector_type;

BEGIN -- select_digits

   IF restarting THEN

      -- Setting up to restart PDI search from checkpoint

      IF (digit_to_place /= smin.radix-1) AND THEN
         (select_vector(digit_to_place+1) /=
         checkpoint.select_vector(digit_to_place+1)) THEN

         -- Return if not placing first digit and previous digit
         -- has not yet been selected the right number of times
         RETURN;

      ELSIF (select_vector(digit_to_place) =
         checkpoint.select_vector(digit_to_place)) AND THEN
         (digit_to_place = checkpoint.digit_to_place) THEN

         -- Have completed setup for restart

         -- Clear restarting flag for normal operation
         restarting := false;

         -- Output state at beginning of restart
         text_io.put_line ("RESTARTING");
         text_io.put_line (divider);
         timekeeper.start_time (t0);
         timekeeper.time_stamp (t0);
         text_io.new_line;
         text_io.put ("Elapsed time before restart: ");
         timekeeper.print_elapsed_time (checkpoint.elapsed_time);
         text_io.new_line;
         text_io.put ("Number of combinations tested before restart: ");
         other_io.natural_io.put (checkpoint.combinations_tested,
            width => 1);
         text_io.new_line;
         print_state (smin, digit_to_place, select_vector);
         text_io.put_line (divider);
         text_io.new_line;

      END IF;
```

```
      ELSE

         -- Beginning or continuing PDI search

         -- Get current time and initialize start time if at start of
         -- search
         timekeeper.get_time (t1);
         IF (t0 = uninitialized_time) THEN
            t0 := t1;
         END IF;

         -- Record checkpoint if time since last checkpoint or start of
         -- search is greater than specified time between checkpoints
         IF (t1-t0 >= checkpoint_interval) THEN
            record_checkpoint (order, search_length, smin, free,
               select_vector, digit_to_place);
            t0 := t1;
         END IF;

      END IF;

      -- Tally additional call to select_digits and, if sufficient calls
      -- have been made, handle requests for user interaction, if any
      calls_to_select_digits := calls_to_select_digits + 1;
      IF (calls_to_select_digits >= current_max_calls_to_select_digits) THEN

         -- Reset tally of calls to select_digits
         calls_to_select_digits := 0;

         -- Handle any user interaction
         synchronize.check_interrupts (flag, interval, max_calls);

         -- Process returned flag
         CASE flag IS

            WHEN checkpoint_change =>

               -- Set new time between checkpoints
               checkpoint_interval := interval;
               synchronize.clear_keyboard;

            WHEN max_calls_change =>

               -- Set new number of calls between checks for user input
               -- request
               current_max_calls_to_select_digits := max_calls;
               synchronize.clear_keyboard;

            WHEN status =>

               -- Output search status
               text_io.new_line;
               text_io.put_line ("STATUS");
               text_io.put_line (divider);
               text_io.put ("Performing order-");
               other_io.natural_io.put (order, width => 1);
               IF (order = search_length) THEN
                  text_io.put (" P");
               ELSE
                  text_io.put (", length-");
                  other_io.natural_io.put (search_length, width => 1);
                  text_io.put (" ");
               END IF;
```

```
            text_io.put ("PDI search in base ");
            other_io.natural_io.put (radix, width => 1);
            text_io.new_line;
            text_io.put ("Checkpoint interval: ");
            other_io.duration_io.put (checkpoint_interval,
               fore =>6, aft => 1);
            text_io.put_line (" seconds");
            text_io.put ("Calls to select_digits between polling" &
               " of keyboard: ");
            other_io.natural_io.put (current_max_calls_to_select_digits,
               width => 1);
            text_io.new_line;
            timekeeper.time_stamp;
            text_io.new_line;
            text_io.put ("Elapsed time: ");
            timekeeper.elapsed_time;
            text_io.new_line;
            text_io.put ("Number of combinations tested: ");
            other_io.natural_io.put (combinations_tested, width => 1);
            text_io.new_line;
            print_state (smin, digit_to_place, select_vector);
            text_io.put_line (divider);
            text_io.new_line;
            synchronize.clear_keyboard;

         WHEN continue =>

            -- Resume search
            synchronize.clear_keyboard;

         WHEN no_interrupt =>

            -- Resume search (no user interaction request)
            null;

      END CASE;

   END IF;

   -- Abandon search if smallest possible PDI has too many digits
   IF (length_of_number(smin) > search_length) THEN
      RETURN;
   END IF;

   -- Copy current digit distribution
   new_select_vector := select_vector;

   IF (digit_to_place = 0) THEN

      -- Selecting number of instances of digit 0

      -- Another complete combination has been tested
      combinations_tested := combinations_tested + 1;

      -- Must select free instances of digit 0
      new_select_vector(digit_to_place) := free;

      -- Test for PDI and report if one has been found
      IF is_pdi(smin, new_select_vector) THEN
         report_pdi(smin);
      END IF;
```

```
            ELSE

               -- Selecting number of instances of some digit other than 0

               IF (free = 0) THEN

                  -- No more digits can be selected; fill out digit combination
                  -- with zeroes
                  FOR digit IN REVERSE 0 .. digit_to_place LOOP
                     new_select_vector(digit) := 0;
                  END LOOP;

                  -- Test for PDI and report if one has been found
                  IF is_pdi(smin, new_select_vector) THEN
                     report_pdi(smin);
                  END IF;

               ELSE

                  -- Additional digits can be selected

                  IF (length_of_number(smin) = search_length) THEN

                     -- Smallest possible PDI is right length; try every
                     -- possible number of instances of the next digit to place
                     FOR count IN REVERSE 0 .. free LOOP
                        new_select_vector(digit_to_place) := count;
                        select_digits (smin + sum_table(count, digit_to_place),
                           free - count, new_select_vector, digit_to_place - 1);
                     END LOOP;

                  ELSIF (length_of_number(smin) < search_length) THEN

                     -- Digits selected in combination so far cannot make a
                     -- sufficiently long PDI; begin trying possible numbers
                     -- of instances of the next digit to place, starting with
                     -- greatest possible number and quitting if some number of
                     -- instances cannot possibly lead to finding a PDI
                     FOR count IN REVERSE 0 .. free LOOP
                        new_smin := smin + sum_table(count, digit_to_place);
                        IF (length_of_number(new_smin + sum_table(free - count,
                           digit_to_place -1)) >= search_length) THEN
                           new_select_vector(digit_to_place) := count;
                           select_digits (new_smin, free - count,
                              new_select_vector, digit_to_place -1);
                        ELSE
                           EXIT;
                        END IF;
                     END LOOP;

                  END IF;

               END IF;

            END IF;

         END select_digits;
```

```
BEGIN -- start_search

   -- Output information about search, including whether it is a PDI or
   -- PPDI search
   text_io.new_line;
   text_io.put ("Begin order-");
   other_io.natural_io.put (order, width => 1);
   IF (order = search_length) THEN
     text_io.put (" P");
   ELSE
     text_io.put (", length-");
     other_io.natural_io.put (search_length, width => 1);
     text_io.put (" ");
   END IF;
   text_io.put ("PDI search in base ");
   other_io.natural_io.put (radix, width => 1);
   text_io.new_line;

   -- Output current date and time
   timekeeper.start_time (t);
   timekeeper.time_stamp (t);
   text_io.new_line (2);

   -- Perform initializations for search
   initialize_table (radix, order);
   smin := make_number(radix, 0);
   combinations_tested := 0;

   -- Perform actual search
   select_digits (smin, search_length, select_vector, radix - 1);

   -- Output time of search
   text_io.put ("End search after ");
   timekeeper.elapsed_time;
   text_io.new_line;

   -- Output number of combinations tested
   text_io.put ("Number of combinations tested: ");
   other_io.natural_io.put (combinations_tested, width => 1);
   text_io.new_line (2);

END start_search;
```

```
   BEGIN -- do_search_type

      LOOP

         SELECT

            -- Wait for call to perform PDI search
            ACCEPT enter_start_search (radix : IN radix_range;
               order : IN natural; search_length : IN digit_range) DO

               -- Perform search
               start_search (radix, order, search_length);

            END enter_start_search;

         OR

            -- Wait for call to restart search from checkpoint
            ACCEPT enter_start_search_from_checkpoint;

            BEGIN

               -- Get search parameters from  checkpoint file
               checkpoint_io.open (file => checkpoint_file,
                  mode => checkpoint_io.in_file, name => "pdi.ckp");
               checkpoint_io.read (file => checkpoint_file,
                  item => checkpoint);
               checkpoint_io.close (file => checkpoint_file);

            EXCEPTION

               WHEN checkpoint_io.name_error =>
                  text_io.new_line;
                  text_io.put_line ("Cannot find file ""pdi.ckp""");
                  RAISE file_open_error;

               WHEN checkpoint_io.use_error =>
                  text_io.new_line;
                  text_io.put_line ("Cannot open file ""pdi.ckp""");
                  RAISE file_open_error;

               WHEN checkpoint_io.end_error =>
                  text_io.new_line;
                  text_io.put_line ("End-of-file error reading file" &
                     " ""pdi.ckp""");
                  RAISE file_open_error;

               WHEN OTHERS =>
                  text_io.new_line;
                  text_io.put_line ("Problem encountered reading file " &
                     """pdi.ckp""");
                  RAISE file_open_error;

            END;

            -- Perform search
            restarting := true;
            start_search (checkpoint.smin.radix, checkpoint.order,
               checkpoint.search_length);

            -- Abort process to allow termination with TERMINATE option of
            -- Select
            ABORT monitor_keyboard;
```

```
      OR

         TERMINATE;

      END SELECT;

   END LOOP;

EXCEPTION

   WHEN file_open_error =>
      text_io.new_line;
      text_io.put_line ("Terminating program");
      text_io.new_line;
      ABORT monitor_keyboard;

   WHEN constraint_error | numeric_error =>
      text_io.new_line;
      text_io.put_line ("Numerical error encountered");
      text_io.new_line;
      text_io.put_line ("Terminating program");
      text_io.new_line;
      ABORT monitor_keyboard;

   WHEN storage_error =>
      text_io.new_line;
      text_io.put_line("Storage error encountered");
      text_io.new_line;
      text_io.put_line ("Terminating program");
      text_io.new_line;
      ABORT monitor_keyboard;

   WHEN OTHERS =>
      text_io.new_line;
      text_io.put_line("Program error encountered");
      text_io.new_line;
      text_io.put_line ("Terminating program");
      text_io.new_line;
      ABORT monitor_keyboard;

END do_search_type;
```

```
--============================================================================
--|
--| TASK NAME: numbers.search.monitor_keyboard
--|
--| ALGORITHM/STRATEGY: The task waits for a call to entry
--|    begin_keyboard_monitor.  It then enters a loop to alternately read a
--|    line from the keyboard and call synchronize.keyboard_interrupt.
--|
--| NOTES: None.
--|
--============================================================================

WITH text_io;

SEPARATE (numbers.search)

TASK BODY monitor_keyboard IS

   -- Position of last character on input line
   last        : natural;
    --Input line entered by user
   response    : string(1 .. line_length);

BEGIN -- monitor_keyboard

   -- Wait for signal to begin monitoring for user input
   ACCEPT begin_keyboard_monitor;

   -- Process signals from user
   LOOP

      -- Read input
      text_io.get_line (item => response, last => last);

      -- Signal interrupt
      synchronize.keyboard_interrupt;

   END LOOP;

END monitor_keyboard;
```

```
--============================================================================
--|
--| PACKAGE NAME: numbers
--|
--| PURPOSE: Provide multiple-precision integer type and operations.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/20/90
--| DATE OF LAST REVISION: 8/9/90          VERSION: 2.1
--|
--| NOTES: The operations provided are extensive, but not exhaustive.
--|    Uninitialized numbers and numbers resulting from invalid operations are
--|    "invalid," that is, they have a special representation recognized as
--|    invalid.  Zero is always treated as positive.
--|
--============================================================================

PACKAGE numbers          IS

   -- Constants related to the representation of numbers:
   -- Number of digits in representation
   maximum_number_length : CONSTANT := 60;
   -- Largest possible radix (must be at least 2)
   maximum_radix         : CONSTANT := 90;
   -- Maximum number of digits required to represent radix
   maximum_radix_length  : CONSTANT := 2;
   -- Maximum number of digits required to represent single digit of number
   maximum_digit_length  : CONSTANT := 2;

   -- Type derivatives of above constants:
   -- To hold representation of a single digit
   SUBTYPE digit_string  IS string(1 .. maximum_digit_length);
   -- Range of values possible for radix
   SUBTYPE radix_range   IS natural RANGE 1 .. maximum_radix;
   -- Range of digit positions (low-order digit numbered 1)
   SUBTYPE digit_range   IS natural RANGE 1 .. maximum_number_length;
   -- Range of allowable values for individual digits
   SUBTYPE radix_digit   IS natural RANGE 0 .. (maximum_radix - 1);
```

```
-- Declarations to implement variable-length string representing a number:
-- Maximum number of characters in representation of number (allows for
-- sign and base)
maximum_string_length : CONSTANT := 1 + (maximum_digit_length + 1) *
   maximum_number_length + 6 + maximum_radix_length;
-- Range of length of string
SUBTYPE dynamic_string_length_range IS natural
   RANGE 0 .. maximum_string_length;
-- Range of character positions within string (leftmost character numbered 1)
SUBTYPE dynamic_string_index_range  IS natural
   RANGE 1 .. maximum_string_length;
-- Definition of variable-length string
TYPE dynamic_string   IS
   RECORD

      -- Length of string (possibly 0)
      length : dynamic_string_length_range;

      -- Characters of string
      char   : string(dynamic_string_index_range);

   END RECORD;

-- Variable length string representing "Error"
invalid_number_string  : CONSTANT dynamic_string := (5,
   ('E', 'r', 'r', 'o', 'r', others => ' ') );

-- Number type
TYPE number           IS LIMITED PRIVATE;
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers.make_number
--|
--| PURPOSE: Produce a multiple-precision integer, given a radix and the
--|    value of a single-digit number in that radix.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/25/90         VERSION: 1.1
--|
--| PARAMETERS:
--|    radix                      (in) base of number to be created
--|
--|    digit                      (in) value of number (|digit| < radix)
--|
--| RETURNS: Multiple-precision representation of digit in base-radix.
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: Invalid number returned if digit out
--|    of range.  If digit is negative, negative number is returned.
--|
--| NOTES: None.
--|
--============================================================================

FUNCTION make_number (radix : IN radix_range; digit : IN integer)
   RETURN number;
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers."-" (unary minus)
--|
--| PURPOSE: Negate multiple-precision integer.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 8/9/90          VERSION: 1.2
--|
--| PARAMETERS:
--|    a                          (in) number to be negated
--|
--| RETURNS: Multiple-precision representation of negation of a.
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: None.
--|
--============================================================================

FUNCTION "-" (a : IN number) RETURN number;
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers."abs"
--|
--| PURPOSE: Take absolute value of multiple-precision integer.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/25/90
--| DATE OF LAST REVISION: 7/25/90         VERSION: 1.1
--|
--| PARAMETERS:
--|    a                           (in) number
--|
--| RETURNS: Multiple-precision representation of absolute value of a.
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: None.
--|
--============================================================================

FUNCTION "abs" (a : IN number) RETURN number;
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers."<="
--|
--| PURPOSE: Compare multiple-precision integers.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/25/90         VERSION: 1.1
--|
--| PARAMETERS:
--|    a                          (in) number
--|
--|    b                          (in) number
--|
--| RETURNS: Boolean result of a <= b
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: Returns false if either parameter is invalid or
--|    if the parameters are in different bases, irrespective of their actual
--|    values.
--|
--| NOTES: None.
--|
--============================================================================

FUNCTION "<=" (a, b : IN number) RETURN Boolean;
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers."="
--|
--| PURPOSE: Compare multiple-precision integers for equality.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/25/90         VERSION: 1.1
--|
--| PARAMETERS:
--|    a                          (in) number
--|
--|    b                          (in) number
--|
--| RETURNS: Boolean result of a = b.
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: Returns false if either parameter is invalid or
--|    if the parameters are in different bases, irrespective of their actual
--|    values.
--|
--| NOTES: None.
--|
--============================================================================

FUNCTION "=" (a, b : IN number) RETURN Boolean;
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers.convert_to_string
--|
--| PURPOSE: To convert multiple-precision into a printable string.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/25/90         VERSION: 1.1
--|
--| PARAMETERS:
--|    a                          (in) number
--|
--| RETURNS: String representation of the value of parameter a.  For
--|    example, the hexadecimal number -6A4F causes the string
--|    "- 6 10 4 15 Base 16" to be returned.
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: Returns "Error" if parameter a is invalid.
--|
--| NOTES: None.
--|
--============================================================================

FUNCTION convert_to_string (a : IN number) RETURN dynamic_string;
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers."-" (binary minus)
--|
--| PURPOSE: To subtract two multiple-precision integers.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/26/90         VERSION: 2.1
--|
--| PARAMETERS:
--|    a                          (in) first operand
--|
--|    b                          (in) second operand
--|
--| RETURNS: Multiple-precision representation of a - b.
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: Returns invalid number if either parameter is
--|    invalid or if the parameters are in different bases, irrespective
--|    of their actual values.
--|
--| NOTES: None.
--|
--============================================================================

FUNCTION "-" (a, b : IN number) RETURN number;
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers."+" (binary plus)
--|
--| PURPOSE: To add two multiple-precision integers.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/25/90         VERSION: 1.1
--|
--| PARAMETERS:
--|    a                          (in) first operand
--|
--|    b                          (in) second operand
--|
--| RETURNS: Multiple-precision representation of a + b.
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: Returns invalid number if either parameter is
--|    invalid or if the parameters are in different bases, irrespective
--|    of their actual values.  Overflow generates an invalid result.
--|
--| NOTES: None.
--|
--============================================================================

FUNCTION "+" (a, b : IN number) RETURN number;
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers."*"
--|
--| PURPOSE: To multiply a multiple-precision integer by a single-digit
--|    number in the same base.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/25/90         VERSION: 1.1
--|
--| PARAMETERS:
--|    f                          (in) single-digit value in base of a
--|
--|    a                          (in) number
--|
--| RETURNS: Multiple-precision representation of f * a.
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: Returns invalid number if parameter a is invalid,
--|    or if the result generates overflow.
--|
--| NOTES: The value of a may be negative.
--|
--============================================================================

FUNCTION "*" (f : IN radix_digit; a : IN number) RETURN number;
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers.length_of_number
--|
--| PURPOSE: To determine the number of digits in a multiple-precision
--|    integer.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/25/90         VERSION: 1.1
--|
--| PARAMETERS:
--|    a                          (in) number
--|
--| RETURNS: Number of digits in representation of parameter a.
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: Parameter a is assumed to be a valid number.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: If a has the value 0, the function returns 1.
--|
--============================================================================

FUNCTION length_of_number (a : IN number) RETURN digit_range;
```

```
--============================================================================
--|
--| PACKAGE NAME: numbers.search
--|
--| PURPOSE: To perform searches for PPDIs, including interaction with user.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 7/25/90
--| DATE OF LAST REVISION: 8/8/90          VERSION: 1.1
--|
--| NOTES: None.
--|
--============================================================================

PACKAGE search IS

   -- Messages to be passed from synchronize to do_search to indicate what
   -- action has been requested by the user.  The meanings of the values is
   -- as follows:
   --
   --     checkpoint_change        change checkpoint frequency
   --     max_calls_change         change frequency of keyboard polling
   --     status                   display search status
   --     continue                 resume search after user interrupt
   --     no_interrupt             resume search after no user interrupt
   TYPE interrupt_flag_type IS (checkpoint_change, max_calls_change,
      status, continue, no_interrupt);
```

```
--=====================================================================
--|
--| TASK NAME: numbers.search.process_normal_input
--|
--| PURPOSE: To read parameters for PDI searches from input file (i.e.,
--|    to implement "normal" input mode) and initiate searches.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 7/31/90
--| DATE OF LAST REVISION: 7/28/90         VERSION: 1.2
--|
--| INPUT: (From file "input.dat") triples of integers representing
--|    PDI searches to be performed.  "Input.dat" is a normal text file
--|    (i.e., of type text_io.in_file).  Integers should be separated by
--|    blanks or new lines.  After the third integer of a triple is read,
--|    the remainder of the line on which it occurs is discarded.  The
--|    integers represent, respectively, the radix, order, and length
--|    of the PDIs to be searched for.  The numbers must be consistent
--|    with types (and subtypes) numbers.radix_range, natural, and
--|    numbers.digit_range.
--|
--| OUTPUT: (To default file) error messages.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: Errors relating to opening and reading the
--|    input file are trapped and result in error messages being output
--|    and the program's being terminated.
--|
--| NOTES: A triple is not read from the input file until previous
--|    triple have resulted in completed searches.
--|
--=====================================================================

TASK process_normal_input IS
```

```
      --=====================================================================
      --|
      --| ENTRY NAME: numbers.search.process_normal_input.start
      --|
      --| PURPOSE: To accept signal to start reading input from file
      --|    to direct PDI searches.
      --|
      --| NOTES: Only one call to this entry should be made.  (Entry for
      --|    synchronization purposes only.)
      --|
      --=====================================================================

      ENTRY start;

   END process_normal_input;
```

```
--=====================================================================
--|
--| TASK NAME: numbers.search.monitor_keyboard
--|
--| PURPOSE: To read lines from keyboard and call task synchronize.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 7/31/90
--| DATE OF LAST REVISION: 8/1/90          VERSION: 1.2
--|
--| INPUT: (From default file) lines, whose exact contents is ignored.
--|    After each line is read, a call is made to entry
--|    synchronize.keyboard_interrupt.  This task does nothing until
--|    a call is made to entry begin_keyboard_monitor.
--|
--| OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: This task must be terminated with an abort.
--|
--=====================================================================

TASK monitor_keyboard IS
```

```
   --=====================================================================
   --|
   --| ENTRY NAME: numbers.search.monitor_keyboard.begin_keyboard_monitor
   --|
   --| PURPOSE: To accept signal to start reading input lines.
   --|
   --| NOTES: Only one call to this entry should be made.  (Entry for
   --|    synchronization purposes only.)
   --|
   --=====================================================================

   ENTRY begin_keyboard_monitor;

END monitor_keyboard;
```

```
--=====================================================================
--|
--| TASK NAME: numbers.search.synchronize
--|
--| PURPOSE: To coordinate PDI search and user requests from the
--|    keyboard.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 7/31/90
--| DATE OF LAST REVISION: 8/1/90          VERSION: 1.2
--|
--| INPUT: (From default file ) user commands and associated
--|    parameters.  Commands recognized are "r" (resume search), "s"
--|    (display status of search), "c" (change checkpoint interval),
--|    "k" (change keyboard polling frequency), and "q" (quit search).
--|    Commands "c" and "k" require input of some number of seconds
--|    (time between checkpoints) and an integer (some number of
--|    invocations of do_search.start_search.select_digits between
--|    check for user requests).
--|
--| OUTPUT: (To default file) prompts for commands and error messages.
--|
--| ASSUMPTIONS/LIMITATIONS: In order to achieve proper termination,
--|    this task assumes a call is made to entry start_monitor.
--|
--| ERROR CHECKS/RESPONSES: Erroneous user input results in error
--|    messages and reprompts.
--|
--| NOTES: When "q" command is entered, synchronize aborts tasks
--|    do_search, monitor_keyboard, and process_normal_input.  In other
--|    cases, synchronize terminates using a terminate option in select
--|    statement.
--|
--=====================================================================

TASK synchronize IS
```

```
--=====================================================================
--|
--| ENTRY NAME: numbers.search.synchronize.start_monitor
--|
--| PURPOSE: To accept signal to start executing the function of
--|    package synchronize.
--|
--| NOTES: Only one call to this entry should be made.  (Entry for
--|    synchronization purposes only.)  Call results in call to
--|    entry monitor_keyboard.begin_keyboard_monitor to initiate
--|    operation of task monitor_keyboard.
--|
--=====================================================================

ENTRY start_monitor;
```

```
--=====================================================================
--|
--| ENTRY NAME: numbers.search.synchronize.check_interrupts
--|
--| PURPOSE: To allow task synchronize to take user commands and
--|    execute them or have the caller (task do_search) execute them,
--|    that is, to coordinate user command processing with the PDI
--|    search.
--|
--| PARAMETERS:
--|    flag                       (out) message flag returned by
--|                               synchronize to indicate user
--|                               request (see type definition at
--|                               beginning of package for message
--|                               interpretations
--|
--|    interval                   (out) if command was "c" (and
--|                               therefore flag set to
--|                               checkpoint_change), the new
--|                               number of seconds between
--|                               checkpoints; otherwise meaningless
--|
--|    max_calls                  (out) if command was "k" (and
--|                               therefore flag set to
--|                               max_calls_change), the new number
--|                               of calls of
--|                               do_search.start_search.select_digits
--|                               between calls to this entry;
--|                               otherwise meaningless
--|
--| NOTES: When rendezvous completes, task synchronize waits for
--|    confirmation that the caller has acted on the information
--|    transmitted by parameter flag.  This confirmation is in the
--|    form of a call to entry clear_keyboard.  Only when there
--|    was no request for service by the user (that is, when flag is
--|    returned as no_interrupt) is no subsequent call to
--|    clear_keyboard expected.
--|
--=====================================================================

ENTRY check_interrupts (flag : OUT interrupt_flag_type;
   interval : OUT duration; max_calls : OUT natural);
```

```
--=====================================================================
--|
--| ENTRY NAME: numbers.search.synchronize.keyboard_interrupt
--|
--| PURPOSE: To accept signal that user has requested service.
--|
--| NOTES: Rendezvous does not complete until service has been
--|    provided.
--|
--=====================================================================

ENTRY keyboard_interrupt;
```

```
      --=====================================================================
      --|
      --| ENTRY NAME: numbers.search.synchronize.clear_keyboard
      --|
      --| PURPOSE: To receive confirmation that information returned
      --|    from rendezvous at entry check_interrupts has been
      --|    appropriately acted upon.
      --|
      --| NOTES: See notes for entry check_interrupts.
      --|
      --=====================================================================

      ENTRY clear_keyboard;

   END synchronize;
```

```
--=========================================================================
--|
--| TASK TYPE NAME: numbers.search.(do_search_type)
--|
--| PURPOSE: To actually perform PDI searches.  (Only one task is
--|    needed.  A task type is required, so the storage allocated
--|    can be set with FOR ... USE.)
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 7/31/90
--| DATE OF LAST REVISION: 8/9/90          VERSION: 1.3
--|
--| INPUT: (From file "pdi.ckp") if entry
--|    enter_start_search_from_checkpoint is called, task reads
--|    a checkpoint record from this file and restarts a search from
--|    the information in the record.  The task terminates when the
--|    search completes.
--|
--| OUTPUT: (To default file) search-related messages and results.
--|    (To file "pdi.ckp") periodic checkpoint records, from which
--|    a search can be restarted.  The entire file is rewritten each
--|    time a checkpoint record is written, so that the file never
--|    contains more than a single record.
--|
--| ASSUMPTIONS/LIMITATIONS: It is assumed that either entry
--|    enter_start_search_from_checkpoint is called once or entry
--|    enter_start_search is called zero or more times, once for each
--|    search to be performed.
--|
--| ERROR CHECKS/RESPONSES: Problems related to reading the checkpoint
--|    file result in error messages and task termination.
--|
--| NOTES: None.
--|
--=========================================================================

TASK TYPE do_search_type IS
```

```
--=====================================================================
--|
--| ENTRY NAME: numbers.search.(do_search_type).enter_start_search
--|
--| PURPOSE: To tell task to perform a particular PDI search from the
--|    beginning.
--|
--| PARAMETERS:
--|    radix                      (in) base of PDIs sought
--|
--|    order                      (in) order of PDIs sought
--|
--|    search_length              (in) number of digits in PDIs
--|                               sought
--|
--| NOTES: Entry may be called multiple times.
--|
--=====================================================================

ENTRY enter_start_search (radix : IN radix_range;
   order : IN natural; search_length : IN digit_range);
```

```
   --=====================================================================
   --|
   --| ENTRY NAME: numbers.search.(do_search_type).-
   --|    enter_start_search_from_checkpoint
   --|
   --| PURPOSE: To receive signal to begin PDI search using checkpoint
   --|    record in "pdi.ckp".
   --|
   --| NOTES: This entry should be called at most once.
   --|
   --=====================================================================

   ENTRY enter_start_search_from_checkpoint;

END do_search_type;
```

```
        -- Assure adequate storage is available for search; the number below
        -- is quite machine-dependent
        FOR do_search_type'storage_size USE 1_048_576;

        -- Actual task to perform PDI searches
        do_search : do_search_type;

    END search;
```

```
PRIVATE

   -- Flag used to indicate invalid number
   invalid_number_flag    : CONSTANT := 1;
   -- Sign values
   TYPE sign_type         IS (plus, minus);
   -- To represent digits of number
   TYPE digit_set         IS ARRAY(digit_range) OF radix_digit;
   -- Representation of number
   TYPE number            IS
      RECORD

         -- Base (initialized to invalid value)
         radix            : radix_range := invalid_number_flag;

         -- Position of leftmost digit
         high_order_digit : digit_range;

         -- Sign of number
         sign             : sign_type;

         -- Digits of number
         digit            : digit_set;

      END RECORD;

END numbers;
```

```
--============================================================================
--|
--| PACKAGE NAME: numbers
--|
--| NOTES: Operators generally assume multiple-precision parameters to be
--|    canonical--to be valid representations or to be "invalid," that is, to
--|    carry an invalid flag.  Except for zero itself, canonical numbers have
--|    no leading 0 digits.  Numbers returned are intended to be canonical
--|    also.  Zero is always represented as positive.
--|
--============================================================================

PACKAGE BODY numbers IS

   -- Range of possible carries or borrows
   SUBTYPE carry_or_borrow IS radix_digit RANGE 0 .. 1;
```

```
--=============================================================================
--|
--| FUNCTION NAME: numbers.make_number
--|
--| ALGORITHM/STRATEGY: Check parameter validity and set values of result.
--|
--| NOTES: None.
--|
--=============================================================================

FUNCTION make_number (radix : IN radix_range; digit : IN integer)
   RETURN number IS

   -- Number to be returned
   result : number;

BEGIN -- make_number

   -- Check if parameters valid
   IF (ABS(digit) < radix) THEN

      -- Set sign
      IF (digit < 0) THEN
         result.sign := minus;
      ELSE
         result.sign := plus;
      END IF;

      -- Set radix, length of number, and single digit
      result.radix := radix;
      result.high_order_digit := 1;
      result.digit(1) := ABS(digit);

   ELSE

      -- Flag number as invalid
      result.radix := invalid_number_flag;

   END IF;

   -- Return number generated
   RETURN result;

END make_number;
```

```
--=============================================================================
--|
--| FUNCTION NAME: numbers."-" (unary minus)
--|
--| ASSUMPTIONS/LIMITATIONS: Parameter a is assumed canonical.
--|
--| ALGORITHM/STRATEGY: Copy number and reverse sign unless parameter is 0.
--|
--| NOTES: None.
--|
--=============================================================================

FUNCTION "-" (a : IN number) RETURN number IS

   -- Number to be returned
   result : number;

BEGIN -- "-"

   -- Copy number
   result := a;

   IF (a.high_order_digit = 1) AND THEN (a.digit(1) = 0) THEN

      -- Set sign to positive for zero a
      result.sign := plus;

   ELSE

      -- Reverse sign for non-zero a
      IF (a.sign = plus) THEN
         result.sign := minus;
      ELSE
         result.sign := plus;
      END IF;

   END IF;

   -- Return number generated
   RETURN result;

END "-";
```

```
--=============================================================================
--|
--| FUNCTION NAME: numbers."abs"
--|
--| ASSUMPTIONS/LIMITATIONS: Parameter a is assumed canonical.
--|
--| ALGORITHM/STRATEGY: Set sign of result to plus.
--|
--| NOTES: None.
--|
--=============================================================================

FUNCTION "abs" (a : IN number) RETURN number IS

   -- Number to be returned
   result : number;

BEGIN -- "abs"

   -- Make sign of result positive
   result := a;
   result.sign := plus;

   -- Return number generated
   RETURN result;

END "abs";
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers."<="
--|
--| ASSUMPTIONS/LIMITATIONS: Parameters are assumed canonical.
--|
--| ALGORITHM/STRATEGY: Determine signs of parameters; result is
--|    immediately determined if they differ.  Otherwise, the result is
--|    determined from the number with the most number of digits.  If
--|    signs and lengths are the same, procedure distinguish is used to
--|    determine the result.
--|
--| NOTES: None.
--|
--============================================================================

FUNCTION "<=" (a, b : IN number) RETURN Boolean IS
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers."<=".distinguish
--|
--| PURPOSE: To determine if |x| <= |y|.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 8/9/90          VERSION: 1.2
--|
--| PARAMETERS:
--|    x                          (in) number
--|
--|    y                          (in) number
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: The parameters are assumed to have the same
--|    radix and the same number of digits.
--|
--| ALGORITHM/STRATEGY: Corresponding digits of the parameters are
--|    compared from left to right until a difference between them is
--|    found or it is determined that there is no difference between them.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: The signs of the parameters are ignored.
--|
--============================================================================

FUNCTION distinguish (x, y : IN number) RETURN Boolean IS
```

```
      -- Difference found in corresponding digits
      distinguished : Boolean := false;
      -- Value of x found to be or assumed to be less than that of y
      x_less        : Boolean := false;

   BEGIN -- distinguish

      -- Examine corresponding digits until a difference is found or
      -- all digits are examined
      FOR position IN REVERSE 1 .. x.high_order_digit LOOP

         IF (x.digit(position) > y.digit(position)) THEN

            -- Smaller digit found in y
            distinguished := true;
            x_less := false;

         ELSIF (x.digit(position) < y.digit(position)) THEN

            -- Smaller digit found in x
            distinguished := true;
            x_less := true;

         END IF;

         -- Terminate loop early if unequal corresponding digits found
         EXIT WHEN distinguished;

      END LOOP;

      -- Return true if smaller digit found in x or all digits
      -- are the same
      RETURN (x_less OR ELSE (NOT distinguished));

   END distinguish;
```

```
BEGIN -- "<="

   IF (a.radix /= b.radix) OR ELSE
      (a.radix = invalid_number_flag) OR ELSE
      (b.radix = invalid_number_flag) THEN

      -- Comparison invalid
      RETURN false;

   ELSE

      IF (a.sign /= b.sign) THEN

         -- Signs differ; true if a negative
         RETURN (a.sign = minus);

      ELSE

         If (a.sign = plus) THEN

            --Both signs positive; true if a <= b
            IF (a.high_order_digit < b.high_order_digit) THEN
               RETURN true;
            ELSIF (b.high_order_digit < a.high_order_digit) THEN
               RETURN false;
            ELSE
               RETURN distinguish(a, b);
            END IF;

         ELSE

            --Both signs negative; true if |a| <= |b|
            IF (b.high_order_digit < a.high_order_digit) THEN
               RETURN true;
            ELSIF (a.high_order_digit < b.high_order_digit) THEN
               RETURN false;
            ELSE
               RETURN distinguish(b, a);
            END IF;

         END IF;

      END IF;

   END IF;

END "<=";
```

```
--=============================================================================
--|
--| FUNCTION NAME: numbers."="
--|
--| ASSUMPTIONS/LIMITATIONS: Parameters are assumed canonical.
--|
--| ALGORITHM/STRATEGY: Check radices, sign, and corresponding slices
--|    of digit arrays for inequality.
--|
--| NOTES: None.
--|
--=============================================================================

FUNCTION "=" (a, b : IN number) RETURN Boolean IS

BEGIN -- "="

   IF (a.radix /= b.radix) OR ELSE
      (a.radix = invalid_number_flag) OR ELSE
      (b.radix = invalid_number_flag) OR ELSE
      (a.sign /= b.sign) OR ELSE
      (a.high_order_digit /= b.high_order_digit) THEN

      -- Comparison is not valid, signs differ, or numbers are
      -- of different length
      RETURN false;

   ELSE

      IF (a.digit(1 .. a.high_order_digit) /=
         b.digit(1 .. b.high_order_digit)) THEN

         -- Signs and lengths are same, but digits differ
         RETURN false;

      ELSE

         -- Signs, lengths, and digits are the same
            RETURN true;

      END IF;

   END IF;

END "=";
```

```
--=============================================================================
--|
--| FUNCTION NAME: numbers.convert_to_string
--|
--| ASSUMPTIONS/LIMITATIONS: Parameter is assumed canonical.
--|
--| ALGORITHM/STRATEGY: Character string is generated from left to right.
--|    Attributes are used to generate and space characters.
--|
--| NOTES: None.
--|
--=============================================================================

FUNCTION convert_to_string (a : IN number) RETURN dynamic_string IS

   -- String in which character representation of a is generated
   result        : dynamic_string;
   -- Position in result.char up to which character array has been filled
   string_length : dynamic_string_index_range;

BEGIN -- convert_to_string

   IF (a.radix = invalid_number_flag) THEN

      -- "Error" returned if number invalid
      result := invalid_number_string;

   ELSE

      -- Generate sign
      IF (a.sign = plus) THEN
         result.char(1) := '+';
      ELSE
         result.char(1) := '-';
      END IF;

      -- Generate digits from left to right
      string_length := 1;
      FOR position IN REVERSE 1 .. a.high_order_digit LOOP
         result.char(string_length+1 ..
            string_length+radix_digit'image(a.digit(position))'last) :=
            radix_digit'image(a.digit(position));
         string_length := string_length +
            radix_digit'image(a.digit(position))'last;
      END LOOP;

      -- Generate "Base" and radix
      result.char(string_length+1 .. string_length+5+
         radix_range'image(a.radix)'last) := " Base" &
         radix_range'image(a.radix);
      result.length := string_length + 5 + radix_range'image(a.radix)'last;

   END IF;

   -- Return string generated
   RETURN result;

END convert_to_string;
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers."-" (binary minus)
--|
--| ASSUMPTIONS/LIMITATIONS: Parameters are assumed canonical.
--|
--| ALGORITHM/STRATEGY: If operands are valid, the signs are checked to see
--|    if they are the same.  If they are, the difference is computed using
--|    numbers."+" and numbers."-" (unary minus).  Otherwise, the operand
--|    of larger magnitude is found, and procedure perform_subtraction
--|    is used to find the digits of the difference.
--|
--| NOTES: None.
--|
--============================================================================

FUNCTION "-" (a, b : IN number) RETURN number IS

   -- Number where difference is computed
   result   : number;
```

```
--=========================================================================
--|
--| PROCEDURE NAME: numbers."-".perform_subtraction
--|
--| PURPOSE: To generate the digits of the difference of two
--|    multiple-precision integers.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/26/90         VERSION: 1.1
--|
--| PARAMETERS:
--|    x                          (in) first operand
--|
--|    y                          (in) second operand
--|
--|    result                     (in out) on exit, contains digits and
--|                               pointer to high-order-digit of
--|                               difference; other fields are
--|                               unchanged
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: The operands are assumed to have
--|    legitimate values and to share the same radix.  The magnitude
--|    of the first operand is assumed to be at least as large as that
--|    of the second.
--|
--| ALGORITHM/STRATEGY: If the digits of the operands are the same,
--|    zero is returned.  Otherwise, subtraction is performed on the
--|    meaningful slice of the digit array of the second operand and
--|    the corresponding slice from the first operand, noting any
--|    borrow in the high-order position.  The remaining meaningful
--|    slice from the first operand is copied into the difference, and
--|    the borrow, if any, is propagated.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: The sign and radix fields of parameter result may contain
--|    significant values.  The procedure leaves these unchanged.
--|    The signs and radices of the operands are ignored.
--|
--=========================================================================

PROCEDURE perform_subtraction (x, y : IN number; result : IN OUT number)
   IS

   -- Number borrowed from place to left
   borrow : carry_or_borrow;
   -- Digit position being operated upon
   pos    : digit_range;
```

```
        BEGIN -- perform_subtraction

           IF (x.high_order_digit = y.high_order_digit) AND THEN
              (x.digit(1 .. x.high_order_digit) =
              y.digit(1 .. y.high_order_digit)) THEN

                 -- Difference is zero

                 -- Copy zero value
                 result.high_order_digit := 1;
                 result.digit(1) := 0;

           ELSE

              -- Result is non-zero

              -- Subtract digits in common between x and y
              borrow := 0;
              FOR position IN 1 .. y.high_order_digit LOOP
                 IF (x.digit(position) - borrow < y.digit(position)) THEN
                    result.digit(position) := x.radix + x.digit(position) -
                       y.digit(position);
                    borrow := 1;
                 ELSE
                    result.digit(position) := x.digit(position) - borrow -
                       y.digit(position);
                    borrow := 0;
                 END IF;
              END LOOP;

              -- Copy remaining digits to result
              result.digit(y.high_order_digit+1 .. x.high_order_digit) :=
                 x.digit(y.high_order_digit+1 .. x.high_order_digit);

              -- Propagate borrow to high-order digit
              FOR position IN  y.high_order_digit .. x.high_order_digit LOOP
                 IF (borrow = 1) THEN
                    IF (result.digit(position) = 0) THEN
                       result.digit(position) := x.radix - 1;
                    ELSE
                       result.digit(position) := result.digit(position) - 1;
                       borrow := 0;
                    END IF;
                 ELSE
                    EXIT;
                 END IF;
              END LOOP;

              -- Eliminate leading zeroes
              pos := x.high_order_digit;
              WHILE (result.digit(pos) = 0) LOOP
                 pos := pos - 1;
              END LOOP;
              result.high_order_digit := pos;

           END IF;

        END perform_subtraction;
```

```
BEGIN -- "-"

   IF (a.radix = b.radix) AND THEN
      (a.radix /= invalid_number_flag) AND THEN
      (b.radix /= invalid_number_flag) THEN

      --Subtraction is possible; compute difference
      IF (a.sign = b.sign) THEN

         -- Signs of operands equal
         --
         -- Set radix
         result.radix := a.radix;

         IF (abs(a) <= abs(b)) THEN

            -- Since |b| >= |a|, compute digits of |b| - |a|
            perform_subtraction(b, a, result);

            -- Set sign (special case for zero)
            IF (a.sign = plus) THEN
               IF (result.high_order_digit = 1) AND THEN
                  (result.digit(1) = 0)
                  THEN
                  result.sign := plus;
               ELSE
                  result.sign := minus;
               END IF;
            ELSE
               result.sign := plus;
            END IF;

         ELSE

            -- Since |a| > |b|, compute digits of |a| - |b|
            perform_subtraction(a, b, result);

            -- Set sign
            IF (a.sign = plus) THEN
               result.sign := plus;
            ELSE
               result.sign := minus;
            END IF;

         END IF;

      ELSE

         -- Signs of operands different
         --
         -- Compute difference using addition
         result := a + (-b);

      END IF;

   END IF;

   --- Return difference or invalid number
   RETURN result;

END "-";
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers."+" (binary plus)
--|
--| ASSUMPTIONS/LIMITATIONS: Parameters are assumed canonical.
--|
--| ALGORITHM/STRATEGY: If the operands are valid, the signs of the
--|    operators are compared.  If they are different, the sum is computed
--|    using numbers."abs" and numbers."-" (binary subtraction).  If the
--|    signs are different, the operand with the larger number of digits
--|    is found, and procedure perform_addition is used to compute the digits
--|    of the sum.
--|
--| NOTES: None.
--|
--============================================================================

FUNCTION "+" (a, b : IN number) RETURN number IS

   -- Number where sum is computed
   result : number;
```

```
--============================================================================
--|
--| PROCEDURE NAME: numbers."+".perform_addition
--|
--| PURPOSE: To generate the digits of the sum of two multiple-precision
--|    integers.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/26/90         VERSION: 2.1
--|
--| PARAMETERS:
--|    x                          (in) first operand
--|
--|    y                          (in) second operand
--|
--|    result                     (in out) on exit, contains digits and
--|                               pointer to high-order-digit of
--|                               sum; other fields are
--|                               unchanged
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: The operands are assumed to have
--|    legitimate values and to share the same radix.  The first
--|    operand is assumed to have at least as many digits as that
--|    of the second.
--|
--| ALGORITHM/STRATEGY: The largest digit array slices of meaningful
--|    digits are added, and remaining digits of the first operand
--|    are copied to the sum.  The carry from the first operation is
--|    then propagated.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: The sign and radix fields of parameter result may contain
--|    significant values.  The procedure leaves these unchanged.
--|    The signs and radices of the operands are ignored.
--|
--============================================================================

PROCEDURE perform_addition (x, y : IN number; result : IN OUT number) IS
```

```
      -- Range for addition of two digits
      SUBTYPE digit_sum IS natural RANGE 0 .. (2*maximum_radix - 1);

      -- Number carried from place to right
      carry  : carry_or_borrow;
      -- Sum of corresponding digits
      sum    : digit_sum;

   BEGIN -- perform_addition

      -- Add digits in common between x and y
      carry := 0;
      FOR position IN 1 .. y.high_order_digit LOOP
         sum := x.digit(position) + y.digit(position) + carry;
         IF sum >= x.radix THEN
            result.digit(position) := sum - x.radix;
            carry := 1;
         ELSE
            result.digit(position) := sum;
            carry := 0;
         END IF;
      END LOOP;

      -- Copy remaining digits of x to result
      result.digit(y.high_order_digit+1 .. x.high_order_digit) :=
         x.digit(y.high_order_digit+1 .. x.high_order_digit);

      -- Propagate carry to high-order digit
      FOR position IN y.high_order_digit+1 .. x.high_order_digit LOOP
         IF (carry = 1) THEN
            sum := result.digit(position) + carry;
            IF sum >= x.radix THEN
               result.digit(position) := sum - x.radix;
               carry := 1;
            ELSE
               result.digit(position) := sum;
               carry := 0;
            END IF;
         ELSE
            EXIT;
         END IF;
      END LOOP;

      -- Handle carry from high-order digit, if any
      IF (carry = 1) THEN
         IF (x.high_order_digit < maximum_number_length) THEN
            result.high_order_digit := x.high_order_digit + 1;
            result.digit(x.high_order_digit + 1) := carry;
         ELSE
            result.radix := invalid_number_flag;
         END IF;
      ELSE
         result.high_order_digit := x.high_order_digit;
      END IF;

   END perform_addition;
```

```
BEGIN -- "+"

   IF (a.radix = b.radix) AND THEN
      (a.radix /= invalid_number_flag) AND THEN
      (b.radix /= invalid_number_flag) THEN

      -- Addition is possible; compute sum
      IF (a.sign = b.sign) THEN

         -- Signs of operands equal
         --
         -- Set sign and radix
         result.sign := a.sign;
         result.radix := a.radix;

         IF (a.high_order_digit >= b.high_order_digit) THEN
            perform_addition(a, b, result);
         ELSE
            perform_addition(b, a, result);
         END IF;

      ELSE

         -- Signs of operands different
         --
         -- Compute sum using subtraction
         IF (a.sign = plus) THEN
            result := a - abs(b);
         ELSE
            result := b - abs(a);
         END IF;

      END IF;

   END IF;

   -- Return sum or invalid number
   RETURN result;

END "+";
```

```
--============================================================================
--|
--| FUNCTION NAME: numbers."*"
--|
--| ASSUMPTIONS/LIMITATIONS: Parameter a is assumed canonical.
--|
--| ALGORITHM/STRATEGY: If operands are valid, multiplication is performed.
--|    A zero result is handled as a special case.  In the general case,
--|    the sign and base are determined by the multiple-precision factor.
--|    The product is computed from right to left, and overflow is checked
--|    for before any carry is propagated.
--|
--| NOTES: None.
--|
--============================================================================

FUNCTION "*" (f : IN radix_digit; a : IN number) RETURN number IS

   -- Range for single digit product
   SUBTYPE digit_product     IS natural RANGE 0 .. a.radix**2 - a.radix;
   -- Range for digits in base of operation
   SUBTYPE local_radix_digit IS radix_digit RANGE 0 .. a.radix -1;

   -- Carry from multiplication of one digit
   carry   : local_radix_digit;
   -- Result of single digit multiplication plus carry
   product : digit_product;
   -- Number where product is computed
   result  : number;
```

```
BEGIN -- "*"

   IF (a.radix /= invalid_number_flag AND f<a.radix) THEN

      -- Operands valid; compute product
      --
      -- Set radix
      result.radix := a.radix;
      IF (f=0 OR ELSE (a.high_order_digit=1 AND a.digit(1)=0)) THEN

         -- Set product to zero
         result.high_order_digit := 1;
         result.sign := plus;
         result.digit(1) := 0;

      ELSE

         -- Sum is non-zero
         --
         -- Set sign
         If (a.sign = plus) THEN
            result.sign := plus;
         ELSE
            result.sign := minus;
         END IF;

         -- Multiply by factor, saving high-order carry
         carry := 0;
         FOR position IN 1 .. a.high_order_digit LOOP
            product := (f * a.digit(position)) + carry;
            result.digit(position) := product REM a.radix;
            carry := product/a.radix;
         END LOOP;

         -- Process carry; set product to invalid if overflow occurs
         IF (carry = 0) THEN
            result.high_order_digit := a.high_order_digit;
         ELSE
            IF (a.high_order_digit = maximum_number_length) THEN
               result.radix := invalid_number_flag;
            ELSE
               result.high_order_digit := a.high_order_digit + 1;
               result.digit(result.high_order_digit) := carry;
            END IF;
         END IF;

      END IF;

   END IF;

   -- Return product or invalid number
   RETURN result;

END "*";
```

```
    --============================================================================
    --|
    --| FUNCTION NAME: numbers.length_of_number
    --|
    --| ASSUMPTIONS/LIMITATIONS: Parameter a is assumed canonical.
    --|
    --| ALGORITHM/STRATEGY: Proper value is copied form high-order digit
    --|    digit field of input parameter.
    --|
    --| NOTES: None.
    --|
    --============================================================================

    FUNCTION length_of_number (a : IN number) RETURN digit_range IS

    BEGIN -- length_of_number

       -- Return length from high-order digit field
       RETURN (a.high_order_digit);

    END length_of_number;

    PACKAGE BODY search IS SEPARATE;

  END numbers;
```

```
--============================================================================
--|
--| PACKAGE NAME: other_io
--|
--| PURPOSE: To provide instantiated I/O packages for various units.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/20/90
--| DATE OF LAST REVISION: 8/1/90          VERSION: 1.1
--|
--| NOTES: This package required by packages numbers.search and timekeeper.
--|
--============================================================================

WITH text_io;

PACKAGE other_io IS

   PACKAGE duration_io IS NEW text_io.fixed_io   (duration);
   PACKAGE natural_io  IS NEW text_io.integer_io (natural);
   PACKAGE integer_io  IS NEW text_io.integer_io (integer);

END other_io;
```

```
--================================================================================
--|
--| PROCEDURE NAME: pdi (main procedure)
--|
--| PURPOSE: To find PDIs and PPDIs
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/24/90
--| DATE OF LAST REVISION: 8/8/90          VERSION: 2.0
--|
--| INPUT: (From default file) commands and requests to accept commands from
--|    user.
--|
--|    (From file "pdi.ckp") checkpoint information.  File contains at most
--|    one checkpoint record.
--|
--| OUTPUT: (To default file) prompts, status information, and PDIs found.
--|
--|    (To file "pdi.ckp") information written by program to all restart
--|    from a checkpoint.
--|
--| ASSUMPTIONS/LIMITATIONS: The program has certain built-in limitations
--|    that can be altered by changing constants in the code:
--|
--|              1. Numbers longer than 60 digits cannot be handled.
--|                 Therefore, searches can be performed for at most
--|                 length-60 PDIs.
--|
--|              2. The maximum radix that can be accommodated is 90.
--|
--|    Searches for long-length PDIs may cause two kinds of problems:
--|
--|              1. No provision is made for output lines exceeding the line
--|                 length normally handled by the output device.
--|
--|              2. Available storage may be insufficient for performing some
--|                 searches.  The amount needed is highly system-dependent,
--|                 and the storage allocated for the search task may need
--|                 to be adjusted for particular systems or particular
--|                 searches.
--|
--| ALGORITHM/STRATEGY: Let the radix of the PDIs being sought be r.  (See
--|    background notes below.)  The search proceeds by selecting the number
--|    of digit r-1 in the number to be tested, then the number of digit
--|     r-2, etc.  A combination of digits is counted as tested whenever
--|    the distribution of all digits r-1, r-2, ... , 0 is determined.
--|    Each time procedure select_digits is called, the program
--|    selects the distribution of another digit.  The program selects
--|    the maximum number of occurrences of a digit before a distribution
--|    involving fewer occurrences is tried.  As the number of instances of
--|    each digit is determined, the sum of these digits raised to the
--|    search-order power is computed in smin (the minimum summation value
--|    of the distribution being computed).  Should smin become a number
--|    whose length is greater than the length of the PDIs being sought,
--|    the program backtracks, thereby removing large digits in favor of
--|    smaller ones.  A one-dimensional array, select_vector, keeps track of
--|    how many instances of each digit have been selected in the current
--|    distribution.  The search algorithm finds PDIs in roughly reverse
--|    numerical order.
```

```
--| ERROR CHECKS/RESPONSES: Most errors likely to occur are caught by the
--|    program and result in error messages written to the output file.
--|    Some of the these messages are rather nonspecific, however, and
--|    may require the user to make reasonable inferences.  It is intended
--|    that invalid input from all sources (including missing and unreadable
--|    files) be recognized as erroneous and that appropriate user error
--|    messages be displayed.
--|
--| COPYRIGHT NOTICE:
--|
--|    Copyright (c) 1990 by Carnegie Mellon University, Pittsburgh, Pa.
--|
--|    Distribution: Approved for public release; distribution is unlimited.
--|
--|    Produced by the Software Engineering Institute (SEI).  The SEI is a
--|    federally funded research and development center operated by Carnegie
--|    Mellon University and sponsored by the U.S. Department of Defense under
--|    contract F19628-90-C-0003.
--|
--|    Permission to make copies or derivative works of this software is
--|    granted, without fee, provided that the copies, derivative works, and
--|    supporting documentation are not made or distributed for direct
--|    commercial advantage, and that all copies, derivative works, and
--|    supporting documentation contain this copyright notice and state that
--|    copying is by permission of Carnegie Mellon University.
--|
--|
--| NOTES:
--|
--|    1. Background
--|
--|    This program finds perfect digital invariants (PDIs) and
--|    pluperfect digital invariants (PPDIs).  A PDI is an integer, the
--|    sum of whose digits, each raised to the same integral power, equals
--|    the number itself.  For example,
--|
--|                     5    5    5    5
--|           4150 = 4  + 1  + 5  + 0  = 1024 + 1 + 3125 + 0.
--|
--|    We call 4150 an order-5 (for the exponent), length-4 (for the number of
--|    digits) PDI.  Being a PDI is a property of the number and its base
--|    (radix).  It is easy to verify, for example, that 4150, interpreted as
--|    an octal (base-8) number, is not an order-5 PDI.  On the other hand,
--|    the octal number 4423 is an order-5 PDI because
--|
--|                     5    5    5    5
--|           4423 = 4  + 4  + 2  + 3  = 2000 + 2000 + 40 + 363,
--|
--|    where all the arithmetic shown is in octal.  Because the radix of the
--|    number is significant, we should speak of 4150 as an order-5,
--|    length-4, base-10 PDI and of 4423 as an order-5, length-4, base-8 PDI.
--|    (Abbreviations are possible; we might say 4150 is an order-5 PDI,
--|    understanding that the base is 10 and assuming the length is apparent.)
--|
--|    A PDI whose order is the same as its length is known as a pluperfect
--|    digital invariant, or PPDI.  The decimal number 8208, for example, is
--|    an order-4 PPDI, since
--|
--|                     4    4    4    4
--|           8208 = 8  + 2  + 0  + 8  = 4096 + 16 + 0 + 4096.
```

```
--|    PDIs and PPDIs have been tabulated, but questions remain about their
--|    properties and distribution.  (It is not known if there are PPDIs of
--|    orders other than 1 in all bases, although there are non-trivial PPDIs
--|    in nearly all bases.)  These are not burning questions of mathematics,
--|    but they have received a degree of attention, particularly from
--|    the recreational mathematics community.  This program performs
--|    exhaustive searches for PDIs and PPDIs.
--|
--|
--|    2. References
--|
--|       Martin Gardner, THE INCREDIBLE DR. MATRIX, pp. 205-209.  New York:
--|          Charles Scribner's Sons, 1976.
--|
--|       Lionel Deimel & Michael Jones, "Finding Pluperfect Digital
--|          Invariants: Techniques, Results and Observations."  JOURNAL OF
--|          RECREATIONAL MATHEMATICS 14:2, pp. 87-107, 1981-1982.
--|
--|
--|    3. Program Operation Overview
--|
--|    The program interacts with the user through the standard input file
--|    (presumed to be the keyboard).  Output is sent to the standard output
--|    file (presumed to be a CRT).  If the user needs to save program output,
--|    some mechanism is likely available through the operating system.
--|    (Most operating systems have a way of capturing all terminal output
--|    in a file.)  The user interactively indicates whether search parameters
--|    (which tell the program what searches to perform) are to be taken from
--|    the file named "input.dat" or whether a previously begun search is to be
--|    restarted from a checkpoint.  In the latter case, checkpoint information
--|    is read from file "pdi.ckp."  Because searches can be long-running,
--|    the program records a checkpoint in "pdi.ckp" each hour.  Should
--|    processing be interrupted for any reason, the program can be restarted
--|    from its state as of the last time a checkpoint was written.
--|
--|    File "input.dat" should contain any number of triples of natural
--|    numbers, each triple specifying a search the program is to carry out.
--|    Numbers may be separated by spaces or line breaks; "normal" input
--|    should have a triple on each line of the file, although this format
--|    is not required.  In any case, any characters on the line after the
--|    third number of a triple are ignored, and the first number of the next
--|    triple is sought on the next line.  The numbers of the triple
--|    represent, respectively, the radix, order, and length of the PDIs being
--|    sought.  The second and third numbers of a triple should be equal,
--|    of course, if the search is for PPDIs.
--|
--|    If the program is started from a checkpoint, the contents of
--|    "input.dat," if any, are ignored; if the interrupted search is
--|    completed, the program terminates.
```

```
--|    During a search, the following is written to the screen:
--|
--|              1. When the search begins, the date, time, and search
--|                 parameters.
--|
--|              2. When a PDI is found, the number, elapsed time since the
--|                 beginning of the search, and the number of combinations
--|                 tested.  The time shown is clock time, not processor time.
--|
--|              3. When a checkpoint record is written, the date, time,
--|                 elapsed time, number of combinations tested, and search
--|                 state information: the values of smin, digit_to_place
--|                 (the next digit whose number of instances is to be
--|                 determined by select_digits), and select_vector.
--|
--|              4. When the search ends, the elapsed time and number of
--|                 combinations tested.
--|
--|    While the program is running, the user may interact with it by typing
--|    <return>.  (Actually, the program reads any line entered, but ignores
--|    the actual contents of the line.)  After a brief delay--for the sake
--|    of efficiency, the program checks the keyboard infrequently--the user
--|    is prompted with:
--|
--|              Enter "r" to resume search,
--|                    "s" for status check of search,
--|                    "c" to change checkpoint interval,
--|                    "k" to change keyboard polling frequency, or
--|                    "q" to quit:
--|
--|    The user should enter the appropriate letter and <return>.  These
--|    commands, respectively, cause the program to resume its search,
--|    display the state of the search, change the frequency
--|    with which checkpoints are taken (from once each hour), change the
--|    frequency with which the keyboard is checked (the default is after
--|    each 3000 calls to procedure select_digits), and terminate the program.
--|    The state information includes all the information printed when a
--|    checkpoint is taken, plus the search parameters, checkpoint interval,
--|    and keyboard polling frequency.
--|
--|    The user should recognize that program efficiency and flexibility are
--|    affected by changing the checkpoint interval and polling frequency.
--|    If checkpoints are recorded too often, the search process is slowed
--|    down, although less computing time is wasted should the program have
--|    to be restarted from a checkpoint.  If the number of calls to
--|    select_digits before the keyboard is checked is made too low, the
--|    program becomes very responsive to keyboard interrupts, but at a high
--|    price, measured in terms of search efficiency.
--|
--|
--|    4. Procedure Overview
--|
--|    Procedure pdi prompts the user to determine the source of input.
--|    Entry calls are made to task search to begin searches.  Illegal responses
--|    by the user cause the prompt to be redisplayed.  After the
--|    appropriate call is made to begin searching, an entry call to
--|    synchronize is made to initiate monitoring of the keyboard while
--|    searches proceed.
--|
--=============================================================================

WITH numbers;
WITH text_io;
```

```
PROCEDURE pdi IS

   -- Maximum length of input line
   line_length : CONSTANT := 80;

   -- Position of last character on input line
   last        : natural;
   -- Flag indicating (if true) that a prompt for a user command must be output
   repeat      : Boolean := true;
    --Input line entered by user
   response    : string(1 .. line_length);

BEGIN -- pdi

   -- Determine input option from user
   WHILE repeat LOOP

      -- Assume no need to reprompt
      repeat := false;

      -- Prompt user and read response
      text_io.new_line;
      text_io.put_line
         ("Enter ""n"" for normal input from file ""input.dat""");
      text_io.put_line
         ("   or ""r"" to restart from file ""checkpoint"": ");
      text_io.get_line (item => response, last => last);

      -- Interpret user response
      IF (last = 1) THEN
         IF (response(1) = 'n') THEN

            -- Take search parameters from "input.dat"
            numbers.search.process_normal_input.start;

         ELSIF (response(1) = 'r') THEN

            -- Begin search from checkpoint file "pdi.ckp"
            numbers.search.do_search.enter_start_search_from_checkpoint;

         ELSE

            -- Invalid user input (unrecognized character); must reprompt
            repeat := true;
            text_io.new_line;

         END IF;

      ELSE

         -- Invalid user input (>1 character entered); must reprompt
         repeat := true;
         text_io.new_line;

      END IF;

   END LOOP;

   --Begin monitoring keyboard for user request to enter commands
   numbers.search.synchronize.start_monitor;

END pdi;
```

```
--============================================================================
--|
--| TASK NAME: numbers.search.process_normal_input
--|
--| ALGORITHM/STRATEGY: Search parameters are read from file and, for each
--|    set, entry do_search.enter_start_search is called.
--|
--| NOTES: None.
--|
--============================================================================

WITH other_io;
WITH text_io;

SEPARATE (numbers.search)

TASK BODY process_normal_input IS

   -- Input file
   input         : text_io.file_type;
   -- Order of PDIs to be sought
   order         : natural;
   -- Radix of PDIs to be sought
   radix         : numbers.radix_range;
   -- Length of PDIs to be sought
   search_length : numbers.digit_range;

   -- Exception raised for any error related to opening input file
   file_open_error : EXCEPTION;

BEGIN -- process_normal_input

   SELECT

      -- Wait for signal to begin reading file
      ACCEPT start;

      BEGIN

          -- Open input file
         text_io.open (file => input, mode => text_io.in_file,
            name => "input.dat");

      EXCEPTION

         WHEN text_io.name_error =>
            text_io.new_line;
            text_io.put_line ("Cannot find file ""input.dat""");
            RAISE file_open_error;

         WHEN text_io.use_error =>
            text_io.new_line;
            text_io.put_line ("Cannot open file ""input.dat""");
            RAISE file_open_error;

         WHEN OTHERS =>
            text_io.new_line;
            text_io.put_line ("Problem encountered opening file " &
               """input.dat""");
            RAISE file_open_error;

      END;
```

```
      -- Process search parameters until end-of-file reached
      WHILE (NOT text_io.end_of_file(input)) LOOP

         -- Read parameters
         other_io.natural_io.get(input, radix);
         other_io.natural_io.get(input, order);
         other_io.natural_io.get(input, search_length);
         text_io.skip_line(file => input);

         -- Call entry to begin search
         search.do_search.enter_start_search (radix, order, search_length);

      END LOOP;

      ABORT monitor_keyboard;

   OR

      TERMINATE;

   END SELECT;

EXCEPTION

   WHEN file_open_error =>
      text_io.new_line;
      text_io.put_line ("Terminating program");
      text_io.new_line;
      ABORT monitor_keyboard;

   WHEN text_io.data_error =>
      text_io.new_line;
      text_io.put_line ("Error reading file ""input.dat""");
      text_io.new_line;
      text_io.put_line ("Terminating program");
      text_io.new_line;
      ABORT monitor_keyboard;

   WHEN text_io.end_error =>
      text_io.new_line;
      text_io.put_line ("End-of-file error reading file ""input.dat""");
      text_io.new_line;
      text_io.put_line ("Terminating program");
      text_io.new_line;
      ABORT monitor_keyboard;

   WHEN OTHERS =>
      text_io.new_line;
      text_io.put_line ("Program error encountered");
      text_io.new_line;
      text_io.put_line ("Terminating program");
      text_io.new_line;
      ABORT monitor_keyboard;

END process_normal_input;
```

```
--============================================================================
--|
--| PACKAGE NAME: numbers.search
--|
--| NOTES: For convenience in development and maintenance, all tasks are
--|    separately compiled.
--|
--============================================================================

SEPARATE (numbers)

PACKAGE BODY search IS

   -- Maximum length of input line
   line_length : CONSTANT := 80;

   TASK BODY process_normal_input IS SEPARATE;

   TASK BODY monitor_keyboard IS SEPARATE;

   TASK BODY synchronize IS SEPARATE;

   TASK BODY do_search_type IS SEPARATE;

END search;
```

```
--============================================================================
--|
--| TASK NAME: numbers.search.synchronize
--|
--| ALGORITHM/STRATEGY: Wait for call to entry start_monitor, then start
--|    task monitor_keyboard.  Main part of task accepts calls to either
--|    (1) keyboard_interrupt, followed by check_interrupts or
--|    (2) check_interrupts.
--|
--| NOTES: Several TERMINATE options on SELECT statements are required to
--|    assure proper termination under any reasonable sequence of events.
--|
--============================================================================

WITH other_io;
WITH text_io;

SEPARATE (numbers.search)

TASK BODY synchronize IS

   -- New time requested between checkpoints
   checkpoint_interval            : duration;
   -- Position of last character on input line
   last                           : natural;
   -- New new number of invocations of do_search.select_digits between checks
   -- for keyboard interrupt from user
   new_max_calls_to_select_digits : natural;
   -- Flag indicating (if true) that a prompt for a user command must be output
   repeat                         : Boolean;
   -- Input line entered by user
   response                       : string(1 .. line_length);
```

```
BEGIN -- synchronize

   -- Wait for signal to begin monitoring keyboard for user commands
   ACCEPT start_monitor;

   -- Signal monitor_keyboard to accept user input
   monitor_keyboard.begin_keyboard_monitor;

   LOOP

      SELECT

         -- If user has requested service, calls are accepted first from
         -- task monitor_keyboard, then from task do_search
         ACCEPT keyboard_interrupt DO

            SELECT

               ACCEPT check_interrupts (flag : OUT interrupt_flag_type;
                  interval : OUT duration; max_calls : OUT natural) DO

                  repeat := true;
                  WHILE repeat LOOP

                     -- Prompt for user command
                     text_io.put_line
                        ("Enter ""r"" to resume search,");
                     text_io.put_line
                        ("      ""s"" for status check of search,");
                     text_io.put_line
                        ("      ""c"" to change checkpoint interval,");
                     text_io.put_line
                        ("      ""k"" to change keyboard polling" &
                        " frequency, or");
                     text_io.put_line ("      ""q"" to quit:");

                     -- Read user response
                     text_io.get_line (item => response, last => last);

                     -- Interpret user command
                     IF (last = 1) THEN
                        IF (response(1) = 'r') THEN

                           -- Resumption of search requested
                           text_io.new_line;
                           flag := continue;
                           repeat := false;

                        ELSIF (response(1) = 's') THEN

                           -- Status check requested
                           flag := status;
                           repeat := false;
```

```
ELSIF (response(1) = 'c') THEN

   -- Checkpoint interval change requested
   LOOP
      BEGIN

         -- Prompt for and read new checkpoint interval
         text_io.put_line
            ("Enter new checkpoint interval in" &
            " seconds and tenths of seconds:");
         other_io.duration_io.get
            (checkpoint_interval);
         text_io.skip_line;
         text_io.new_line;
         EXIT;

      EXCEPTION

         WHEN text_io.data_error =>
            text_io.put_line ("Illegal value");
            text_io.skip_line;

      END;
   END LOOP;
   flag := checkpoint_change;
   interval := checkpoint_interval;
   repeat := false;

ELSIF (response(1) = 'k') THEN

   --Keyboard polling frequency change requested
   LOOP
      BEGIN

         -- Prompt for and read new polling frequency
         text_io.put_line
            ("Enter number of digits handled" &
            " between polling of keyboard:");
         other_io.natural_io.get
            (new_max_calls_to_select_digits);
         text_io.skip_line;
         text_io.new_line;
         EXIT;

      EXCEPTION

         WHEN text_io.data_error =>
            text_io.put_line ("Illegal value");
            text_io.skip_line;

      END;
   END LOOP;
   flag := max_calls_change;
   max_calls := new_max_calls_to_select_digits;
   repeat := false;

ELSIF (response(1) = 'q') THEN

   -- Program termination requested
   ABORT process_normal_input;
   ABORT do_search;
   ABORT monitor_keyboard;
   repeat := false;
   text_io.new_line;
```

```
                        ELSE

                           -- Unrecognized command
                           text_io.new_line;

                        END IF;

                     ELSE

                        -- Unrecognized command
                        text_io.new_line;

                     END IF;

                  END LOOP;

               END check_interrupts;

            OR

               TERMINATE;

            END SELECT;

            SELECT

               -- Wait for signal that operation is complete
               ACCEPT clear_keyboard;

            OR

               TERMINATE;

            END SELECT;

         END keyboard_interrupt;

      OR

         -- If user has not requested service, check from task do_search
         -- comes here
         ACCEPT check_interrupts (flag : OUT interrupt_flag_type;
            interval : OUT duration; max_calls : OUT natural) DO

            -- Reply that no service is requested
            flag := no_interrupt;

         END check_interrupts;

      OR

         TERMINATE;

      END SELECT;

   END LOOP;

END synchronize;
```

```
--============================================================================
--|
--| PACKAGE NAME: timekeeper
--|
--| PURPOSE: To provide facilities for measuring and displaying elapsed time
--|    and current time.  Clients can get current time, reset timer,
--|    output elapsed time since timer was reset, compute elapsed time,
--|    and compare times for equality.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 8/9/90          VERSION: 1.7
--|
--| NOTES: Package intended to provide services without additional functions
--|    of standard package calendar.
--|
--============================================================================

WITH calendar;

PACKAGE timekeeper IS

   -- Export type time from standard package calendar
   SUBTYPE time IS calendar.time;

   -- Export "-" function from standard package calendar
   FUNCTION "-" (x : time; y : time) RETURN duration RENAMES calendar."-";

   -- Export "=" function from standard package calendar
   FUNCTION "=" (x : time; y : time) RETURN Boolean RENAMES calendar."=";

   -- Provide special value of type time to indicate need for special
   -- function and to allow procedure calls without corresponding parameter
   uninitialized_time : CONSTANT time := calendar.time_of(1901, 1, 1);
```

```
--============================================================================
--|
--| PROCEDURE NAME: timekeeper.start_time
--|
--| PURPOSE: Reset timer and return current time.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/23/90         VERSION: 1.2
--|
--| PARAMETERS:
--|    t                          (out) current time
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: Procedure elapsed_time can be called to print the elapsed time
--|    since the timer was last reset.
--|
--============================================================================

PROCEDURE start_time (t : OUT time);
```

```
--============================================================================
--|
--| PROCEDURE NAME: timekeeper.time_stamp
--|
--| PURPOSE: Print time specified by input parameter or current time.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/23/90         VERSION: 1.1
--|
--| PARAMETERS:
--|    t                           (in) time to be printed or value of
--|                                   of uninitialized_time
--|
--| INPUT: None.
--|
--| OUTPUT: (To default file) time of parameter t or, if
--|    t=uninitialized_time, current time.  Month, day, year,
--|    hours, minutes, and seconds are shown
--|    (example: "January 22, 1990 at 12:15:10.25").  Output is
--|    neither preceded nor followed by new lines.
--|
--| ASSUMPTIONS/LIMITATIONS: It is assumed the output will fit on the
--|    current line.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: If parameter is omitted, procedure outputs current time.
--|
--============================================================================

PROCEDURE time_stamp (t : IN time := uninitialized_time);
```

```
--============================================================================
--|
--| PROCEDURE NAME: timekeeper.get_time
--|
--| PURPOSE: Get current time.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 7/23/90         VERSION: 1.1
--|
--| PARAMETERS:
--|    t                          (out) current time
--|
--| INPUT/OUTPUT: None.
--|
--| ASSUMPTIONS/LIMITATIONS: None.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: None.
--|
--============================================================================

PROCEDURE get_time (t : OUT time);
```

```
--============================================================================
--|
--| PROCEDURE NAME: timekeeper.print_elapsed_time
--|
--| PURPOSE: Print interval expressed in days, hours, minutes, and seconds.
--|
--| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
--| DATE OF LAST REVISION: 8/9/90          VERSION: 1.2
--|
--| PARAMETERS:
--|    elapsed_in                 (in) interval to be printed
--|
--| INPUT: None.
--|
--| OUTPUT: (To default file) value of elapsed_in, expressed in days,
--|    hours, minutes, and seconds.  Labeled output is a minimum of 12
--|    characters long and is neither preceded nor followed by new
--|    lines.  (Example: "1 day, 23 hours, 14 minutes, 26.1 seconds")
--|
--| ASSUMPTIONS/LIMITATIONS: Parameter elapsed_in is assumed to have a
--|    positive value.
--|
--| ERROR CHECKS/RESPONSES: None.
--|
--| NOTES: None.
--|
--============================================================================

PROCEDURE print_elapsed_time (elapsed_in : IN duration);
```

```
   --============================================================================
   --|
   --| PROCEDURE NAME: timekeeper.elapsed_time
   --|
   --| PURPOSE: Print elapsed time since timer was last reset (by start_time).
   --|
   --| PROGRAMMER: Lionel Deimel              DATE WRITTEN: 5/21/90
   --| DATE OF LAST REVISION: 7/22/90         VERSION: 1.1
   --|
   --| PARAMETERS:
   --|    t                           (in) end of interval being timed or value
   --|                                   of uninitialized_time
   --|
   --| INPUT: None.
   --|
   --| OUTPUT: (To default file) days, hours, minutes, seconds between time
   --|    timer was last reset, to time t,  or, if t=uninitialized_time, to
   --|    current time.  Format is same as that of print_elapsed_time.
   --|
   --| ASSUMPTIONS/LIMITATIONS: Parameter t assumed to be at least as late
   --|    as when timer was last reset.  Output is unpredictable if start_time
   --|    never called.
   --|
   --| ERROR CHECKS/RESPONSES: None.
   --|
   --| NOTES: Timer is reset by call to start_time.  If parameter omitted,
   --|    elapsed time to current time is printed.
   --|
   --============================================================================

   PROCEDURE elapsed_time (t : IN time := uninitialized_time);

END timekeeper;
```

```
--============================================================================
--|
--| PACKAGE NAME: timekeeper
--|
--| NOTES: None.
--|
--============================================================================

WITH other_io;
WITH text_io;

PACKAGE BODY timekeeper  IS

   -- Ranges for valid numbers of hours and minutes
   SUBTYPE hour_number   IS natural RANGE 0 .. 23;
   SUBTYPE minute_number IS natural RANGE 0 .. 59;

   -- Constants in seconds
   seconds    : CONSTANT duration := 1.0;
   one_day    : CONSTANT duration := 86_400*seconds;
   one_hour   : CONSTANT duration :=  3_600*seconds;
   one_minute : CONSTANT duration :=     60*seconds;

   -- Starting time for timer.
   time_0     :          time     := uninitialized_time;
```

```
--=============================================================================
--|
--| PROCEDURE NAME: timekeeper.start_time
--|
--| ALGORITHM/STRATEGY: Get time from calendar.clock.
--|
--| NOTES: None.
--|
--=============================================================================

PROCEDURE start_time (t : OUT time) IS

BEGIN -- start_time

   -- Reset timer to begin at current time and return time to caller.
   time_0 := calendar.clock;
   t := time_0;

END start_time;
```

```
--============================================================================
--|
--| PROCEDURE NAME: timekeeper.time_stamp
--|
--| ALGORITHM/STRATEGY: Procedure calendar.split used to separate time
--|    to be displayed into month, day, year, and seconds.  Seconds broken
--|    down into hours, minutes, and seconds by repeated subtraction.
--|
--| NOTES: Two spaces are used for printing hours, minutes, and seconds,
--|    even if only a one-digit number is required.  Seconds are printed to
--|    two decimal places.
--|
--============================================================================

PROCEDURE time_stamp (t : IN time := uninitialized_time) IS

   -- Type to facilitate printing of months
   TYPE month_name  IS (anuary, ebruary, arch, pril, ay, une, uly, ugust,
      eptember, ctober, ovember, ecember);

   -- Initial characters of months.  Needed because initial capitals
   -- are desired with other letters in month names lowercase.
   month_initial_char : ARRAY (0 .. 11) OF character := ('J', 'F', 'M',
      'A', 'M', 'J', 'J', 'A', 'S', 'O', 'N', 'D');

   -- Package for month_name I/O
   PACKAGE month_io IS NEW text_io.enumeration_io (month_name);

   -- Time to be printed
   stamp_time : time;

   -- Component values of time to be printed
   year       : calendar.year_number;
   month      : calendar.month_number;
   day        : calendar.day_number;
   hours      : hour_number;
   minutes    : minute_number;
   seconds    : calendar.day_duration;
```

```
BEGIN -- time_stamp

   -- Determine if time t or current time is to be printed
   IF (t = uninitialized_time) THEN
      stamp_time := calendar.clock;
   ELSE
      stamp_time := t;
   END IF;

   -- Break time to be printed into components
   calendar.split (stamp_time, year, month, day, seconds);

   -- Output month
   month := month - 1;
   text_io.put (month_initial_char(month));
   month_io.put(month_name'val(month), set => text_io.lower_case);
   text_io.put (" ");

   -- Output day, year, " at "
   other_io.integer_io.put(day, width => 1);
   text_io.put (", ");
   other_io.integer_io.put(year, width => 4);
   text_io.put (" at ");

   -- Calculate and output number of hours
   hours := 0;
   WHILE (seconds >= one_hour) LOOP
      seconds := seconds - one_hour;
      hours := hours + 1;
   END LOOP;
   other_io.integer_io.put (hours, width => 2);
   text_io.put (":");

    -- Calculate and output number of minutes
   minutes := 0;
   WHILE (seconds >= one_minute) LOOP
      seconds := seconds - one_minute;
      minutes := minutes + 1;
   END LOOP;
   other_io.integer_io.put (minutes, width => 2);
   text_io.put (":");

   -- Output number of seconds
   other_io.duration_io.put (seconds, fore => 2, aft => 2);

END time_stamp;
```

```
--=============================================================================
--|
--| PROCEDURE NAME: timekeeper.get_time
--|
--| ALGORITHM/STRATEGY: Time returned from calendar.clock.
--|
--| NOTES: None.
--|
--=============================================================================

PROCEDURE get_time (t : OUT time) IS

BEGIN -- get_time

   t := calendar.clock;

END get_time;
```

```
--============================================================================
--|
--| PROCEDURE NAME: timekeeper.print_elapsed_time
--|
--| ALGORITHM/STRATEGY: Parameter elapsed_in is assigned to elapsed, and
--|    number of days, etc., determined through repeated subtraction.
--|
--| NOTES: Tests are made to assure grammatical output ("1 hour," not
--|    "1 hours," etc.).
--|
--============================================================================

PROCEDURE print_elapsed_time (elapsed_in : IN duration) IS

   -- Seconds of elapsed_in yet to be accounted for in terms of days, etc.
   elapsed    : duration;

   -- Number of days, hours, minutes, and seconds so far found
   -- in interval elapsed_in
   days       : natural := 0;
   hours      : hour_number := 0;
   minutes    : minute_number := 0;
   seconds    : calendar.day_duration;
```

```
   BEGIN -- print_elapsed_time

      -- Save interval to be printed in elapsed
      elapsed := elapsed_in;

       -- Compute number of days in interval
      WHILE (elapsed >= one_day) LOOP
         elapsed := elapsed - one_day;
         days := days + 1;
      END LOOP;

      -- Compute number of hours in interval
      WHILE (elapsed >= one_hour) LOOP
         elapsed := elapsed - one_hour;
         hours := hours + 1;
      END LOOP;

      -- Compute number of minutes in interval
      WHILE (elapsed >= one_minute) LOOP
         elapsed := elapsed - one_minute;
         minutes := minutes + 1;
      END LOOP;

      -- Output number of days, if any
      IF (days > 0) THEN
         other_io.integer_io.put (days, width => 1);
         IF (days = 1) THEN
            text_io.put (" day, ");
         ELSE
            text_io.put (" days, ");
         END IF;
      END IF;

      -- Output number of hours, if any
      IF (days > 0) OR ELSE (hours > 0) THEN
         other_io.integer_io.put (hours, width => 1);
         IF (hours = 1) THEN
            text_io.put (" hour, ");
         ELSE
            text_io.put (" hours, ");
         END IF;
      END IF;

      -- Output number of minutes, if any
      IF (days > 0) OR ELSE (hours > 0) OR ELSE (minutes > 0) THEN
         other_io.integer_io.put (minutes, width => 1);
         IF (minutes = 1) THEN
            text_io.put (" minute, ");
         ELSE
            text_io.put (" minutes, ");
         END IF;
      END IF;

      -- Output number of seconds
      other_io.duration_io.put (elapsed, fore => 2, aft => 1);
      text_io.put (" seconds");

   END print_elapsed_time;
```

```
   --============================================================================
   --|
   --| PROCEDURE NAME: timekeeper.elapsed_time
   --|
   --| ALGORITHM/STRATEGY: Call print_elapsed_time to output length of
   --|    appropriate interval.
   --|
   --| NOTES: None.
   --|
   --============================================================================

   PROCEDURE elapsed_time (t : IN time := uninitialized_time) IS

   BEGIN -- elapsed_time

      -- Output length of interval from from time_0 to current time or
      -- to time t
      IF (t = uninitialized_time) THEN
         print_elapsed_time (calendar.clock - time_0);
      ELSE
         print_elapsed_time (t - time_0);
      END IF;

   END elapsed_time;

END timekeeper;
```

Diskette Order Form

Machine-readable source code for the Ada program in this report (CMU/SEI-90-EM-3) is available from the SEI either on a 5-1/4" diskette for IBM PCs and PC-compatibles or on a 3-1/2" diskette for the Apple Macintosh. Each diskette also includes ASCII versions of the exercises from the text. To receive the distribution diskette, return this form with $10.00 payment to:

Education Program Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Checks should be made payable to **Carnegie Mellon University**.

Please send: ☐ PC Disk ☐ Macintosh Disk

Send to:

Name _____

Address _____

_____

_____

_____

Tel. No. _____

E-mail _____