NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ITS Ada: AN INTELLIGENT TUTORING SYSTEM FOR THE ADA PROGRAMMING LANGUAGE

by

Lori L. DeLooze

December 1991

Thesis Advisor	Yuh-jeng Lee
Second Reader	Leigh W. Bradbury

Approved for public release; distribution is unlimited.

T260100

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/ AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL *(If Applicable)*	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS *(city, state, and ZIP code)* Monterey, CA 93943-5000	7b. ADDRESS *(city, state, and ZIP code)* Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 6b. OFFICE SYMBOL *(If Applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(city, state, and ZIP code)*	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
ITS Ada: An Intelligent Tutoring System for the Ada Programming Language

12. PERSONAL AUTHOR(S)
DeLooze, Lori L.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(year, month,day)*	15. PAGE COUNT
Master's Thesis	FROM TO	December 1991	356

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS *(continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUBGROUP	Intelligent Tutoring System, Intelligent Computer Aided Instruction, Intelligent Computer Assisted Instruction, Ada Education

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Most tutoring systems are machine dependent. In this thesis, we present an intelligent tutoring system, *ITS Ada*, that exploits the designed portability of Ada. *ITS Ada* possesses full knowledge of Ada as defined in the official language reference manual and consists of four major components. The instructional module is a series of screens presented in an order determined by a topic network that covers the complete set of concepts in the Reference Manual for the Ada Programming Language. Superordinate concepts will be presented only after the prerequisite concepts have been mastered. There are exercise problems associated with each topic. The problems are presented by the diagnostic module in either an expository or interrogatory format, based on the student's mastery level for that concept, as determined by the student module. Solutions to the given problems are checked by parsing the student's response into a meaning list and comparing the results with solutions in the expert module. *ITS Ada* has been tested on three platforms: PC/InterAda, Sun SparcStation/Verdix Ada, and Apple Macintosh/Meridian Ada. We believe it can run on any system with a validated Ada compiler.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Yuh-jeng Lee	22b. TELEPHONE (Include Area Code) (408) 646-2361 22c. OFFICE SYMBOL CS/Le

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted   SECURITY CLASSIFICATION OF THIS PAGE

All other editions are obsolete   Unclassified

Approved for public release; distribution is unlimited.

ITS Ada: AN INTELLIGENT TUTORING SYSTEM FOR THE ADA PROGRAMMING LANGUAGE

by

Lori L. DeLooze
Lieutenant , United States Navy
B.A., University of Colorado, 1985
M.B.A., George Washington University, 1989

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1991

ABSTRACT

Most tutoring systems are machine dependent. In this thesis we present an intelligent tutoring system, *ITS Ada*, that exploits the designed portability of Ada. *ITS Ada* possesses full knowledge of Ada as defined in the official language reference manual and consists of four major components. The instructional module is a series of screens presented in an order determined by a topic network that covers the complete set of concepts in the Reference Manual for the Ada Programming Language. Superordinate concepts will be presented only after the prerequisite concepts have been mastered. There are exercise problems associated with each topic. The problems are presented by the diagnostic module in either an expository or interrogatory format, based on the student's mastery level for that concept, as determined by the student module. Solutions to the given problems are checked by parsing the student's response into a meaning list and comparing the results with solutions in the expert module. *ITS Ada* has been tested on three platforms: PC/InterAda, Sun SparcStation/Verdix Ada, and Apple Macintosh/Meridian Ada. We believe it can be run on any system with a validated Ada compiler.

Thesis
R 29815
C.1

TABLE OF CONTENTS

I. INTRODUCTION 1
A. HISTORY OF Ada 1
B. PROLIFERATION OF Ada 2
C. OBJECTIVES 3
D. ORGANIZATION OF THESIS 4
II. INTELLIGENT TUTORING SYSTEMS 6
A. EVOLUTION OF INTELLIGENT TUTORING SYSTEMS 6
B. RELATED WORK 6
1. LISP Tutor 7
2. BRIDGE 7
3. PROUST 11
4. ADA-TUTR 14
C. EVALUATING INTELLIGENT TUTORING SYSTEMS 15
1. Modeling of Knowledge and Reasoning 15
2. Communication 15
3. Cognitive Processing 15
4. Tutoring 16
D. SUMMARY AND EVALUATIONS 16
III. *ITS Ada*: ARCHITECTURAL DESIGN 19
A. COMPONENTS 19
B. USER 20
C. INSTRUCTIONAL MODULE 21
D. EXPERT MODULE 22
E. STUDENT MODULE 23
F. DIAGNOSTIC MODULE 24
G. INTERACTION BETWEEN MODULES 26

IV. *ITS Ada*: IMPLEMENTATION DETAILS 28
A. INSTRUCTIONAL MODULE 29
1. Instructional Screens 29
2. Editor 32
B. STUDENT MODEL 33
C. EXPERT MODULE 34
D. DIAGNOSTIC MODULE 36
E. HARDWARE PLATFORMS 42
V. SAMPLE SESSIONS WITH *ITS Ada* 43
A. INSTRUCTIONAL MATERIAL 43
B. SCENARIO ONE 50
C. SCENARIO TWO 52
VI. CONCLUSION 55
A. ACCOMPLISHMENTS 55
1. A Practical, Fully Functional Ada Tutor 55
2. An Intelligent Tutoring System 56
3. Implementation in Ada 57
B. FUTURE WORK 57
1. Extended Instructional Materials 57
2. Authoring Module 58
3. Timing Constraints When Solving Problems 58
4. Enhanced Instructional Interface 59
APPENDIX A 60
APPENDIX B 63
APPENDIX C 173
REFERENCES 344
INITIAL DISTRIBUTION LIST 347

TABLE OF FIGURES

Figure 1 BRIDGE Screen for Phase 1....8

Figure 2 BRIDGE Screen for Phase 2....9

Figure 3 BRIDGE Screen for Phase 3....10

Figure 4 Sample Correct Program for PROUST....12

Figure 5 Sample Incorrect Program for PROUST....13

Figure 6 PROUST Output for program in Figure 5....14

Figure 7 Architecture of ITS Ada....19

Figure 8 Implementation Diagram for ITS Ada....28

Figure 9 Screen Manipulation Instructions for ITS Ada....30

Figure 10 Menu Screen from ITS Ada....31

Figure 11 Editor Instruction for Commands in ITS Ada....32

Figure 12 Student's Solution....37

Figure 13 Expert's Solution....37

Figure 14 Expert's Meaning List....38

Figure 15 Student's Meaning List....39

Figure 16 Diagnostic Strategy for Next Presentation....40

Figure 17 Screen for Parts of a Program....44

Figure 18 Screen for Simple Statements....45

Figure 19 Screen for If Statement....46

Figure 20 Screen for Case Statement....47

Figure 21 Screen for Alternatives....48

Figure 22 Screen for Loop Statement....49

Figure 23 Screen for Exit Statement....50

Figure 24 Good Student: Solved the First Problem....51

Figure 25 Good Student: Solved the Second Problem 51
Figure 26 Student Solved the First Problem...52
Figure 27 Student Missed the Second Problem 53
Figure 28 Student Shown the Third Problem and Solution..................................54
Figure 29 Student Solved the Forth Problem...54

I. INTRODUCTION

A. HISTORY OF Ada

In 1974, the Department of Defense (DoD) published a report estimating the future costs of its software development at the horrendous amount of over $3 billion annually. In addition, there were hundreds of languages or dialects being used by the DoD and its contractors, making it difficult to interchange programs and programmers and virtually impossible for effective software maintenance (Sammet, 1986, p. 722).

In 1975, at the request of DoD, Fisher of the Institute for Defense Analysis produced a document called STRAWMAN (DoD, 1975), drafting the requirements for a high-level language to meet the programming needs of the DoD. After comments were received and integrated, a revised document, called WOODENMAN, (DoD, 1975) was issued in late 1975, followed by a third set of requirements labeled TINMAN (DoD, 1986) issued in 1976. There was no existing language that met all the requirements expressly stated in the TINMAN document. However, the DoD concluded that it would be possible to create a programming language to meet these requirements.

The DoD issued contracts to four vendors for preliminary language designs based on an updated set of requirements called IRONMAN, also developed by Fisher (DoD, 1977). In early 1978, each of the four submitted documents was given a color code for identification. These language designs were reviewed and the Red and Green designs were chosen for further language development. The final designs, based on the final set of requirements, called STEELMAN, (DoD,

1978) were sent out for public commentary and the Green version was subsequently chosen and named Ada.

The name Ada was chosen to honor Augusta Ada Lovelace, assistant to Charles Babbage and daughter of the poet, Lord Byron. Charles Babbage, an English mathematician, designed a mechanical calculator that used punched cards for control. Since Lady Lovelace had ideas about the control of this machine and helped develop the instructions for calculating with it, she is considered the first computer programmer (Skansholm, 1988, p. 2). In addition, since the name Ada has no military connotations associated with it, the DoD assumed that the civilian community would be more likely to use it, even for civilian programming projects.

B. PROLIFERATION OF Ada

Most other languages created in the past have had either no control over their early growth and changes, or very strong control. The Department of Defense has tried to work in the middle ground between these two extremes. Their main tools for doing this are the policies of trademark and forbidding either subsets or supersets. The DoD policies regarding validation has also helped control the language.

Ada has been trademarked since 1981. The main purpose for the trademark is to prevent compilers that don't conform to the language standards from being sold as true Ada compilers. A true Ada compiler implements neither a subset nor a superset of requirements. If a compiler implements anything other than the requirements stated in the STEELMAN document, it must be stated. Allowing such differences in any Ada compiler will affect the portability of Ada software, and therefore, the compiler will not be validated.

Although compiler validation is not new, it has been handled in a unique way for Ada. Unlike other language validation processes, Ada validation was planned from the very beginning of language development. To be validated, a compiler must successfully run a complete suite of tests. Validation only measures the level of conformance with the standard. Because performance is not measured by the validation tests, any validated compiler may still have bugs and poor performance (Sammet, 1986, p. 727).

Now that validated Ada compilers are available for a large number of computer systems, the DoD has decreed that all new software development must use Ada (DoD, 1983) as the implementation language. For most services, this includes all types of software: logistics, embedded, business, and scientific. The National Aeronautics and Space Administration has adopted Ada. The Federal Aviation Administration has chosen Ada for developing software on the new air traffic control system. In many universities, Ada is replacing Pascal as the introductory language for structured language programming. Ada is even gathering an expanded following in the commercial marketplace (Riehle, 1989, p. 83).

C. OBJECTIVES

Since Ada is a relatively new language and is only now being introduced to students at the undergraduate level, there are many computer professionals who do not have a working knowledge of the language, although they are familiar with general programming language concepts. A formal course in Ada can take upwards of 40 hours and cost thousands of dollars. Such a course is not only expensive, in terms of both time and money, but also impractical since few professionals can afford to spend so much time away from the office.

There is an urgent need to develop a mechanism to teach Ada to these potential Ada programmers, at a time, pace, place that are convenient and comfortable for them. With the proliferation of computer equipment in our environment today, we believe that the computer itself is the best medium to achieve this goal.

However, there are hundreds of different computers in use. If our product is to be a useful tool, the software should be completely portable among them all. Ada has such portability designed into the language. In fact, a perfectly portable Ada program would, without any change, be compilable by any validated Ada compiler, on any host, and be subsequently executable, with the required behavior on the corresponding target for that compiler (Nissen and Wallis, 1984, p. 4). Therefore, in order to help promote wide adoption and application of Ada, we have developed, in Ada, a fully portable intelligent tutoring system, *ITS Ada*, that teaches the basic knowledge of Ada.

D. ORGANIZATION OF THESIS

Chapter II of this thesis provides an overview of intelligent tutoring systems and reviews four computer based tutors that teach introductory programming. Chapters III and IV discuss the overall composition of *ITS Ada*. Chapter III describes the architectural design of *ITS Ada*. Each component of the system is discussed at the conceptual level. The interrelationships between the components are also presented. Chapter IV explores the implementation details of each of these components and the interactions between them. Chapter V includes a sample dialogue of 3 different levels of students: good, fair, and poor. The response by the system is different for each. Finally, Chapter VI discusses our accomplishments and possible future work. The appendices contain the source

code of *ITS Ada* and the data files needed for the system, as well as a topic network describing the hierarchical structure of instructional materials used by the system.

II. INTELLIGENT TUTORING SYSTEMS

A. EVOLUTION OF INTELLIGENT TUTORING SYSTEMS

Traditional applications in computer-assisted instruction (CAI) were a series of screens manipulated by the user's commands (Uhr, 1969; Suppes, 1967; Woods and Hartley, 1971); but artificial intelligence has changed that. Artificial intelligence (AI) is changing the nature of computing by making computers able to emulate human behaviors that require intelligence. Intelligent computer-assisted instruction (ICAI) is the application of artificial intelligence techniques to computer-assisted instruction (Brown, Burton and DeKleer, 1981; Clancey, 1981; Smith et al, 1975). Applications constructed using ICAI techniques are also known as intelligent tutoring systems (ITS).

For a tutoring system to be considered intelligent, it must pass three tests. (Burns and Capps, 1988, p. 1) First, the subject matter, or domain, must be "known" to the computer system well enough for this embedded expert to draw inferences or solve problems in the domain. Second, the system must be able to deduce a learner's approximation of that knowledge. Third, the tutorial strategy or pedagogy must be intelligent in that the "instructor in the box" can implement strategies to reduce the difference between expert and student performance.

B. RELATED WORK

We present the background of three Intelligent Tutoring Systems which are targeted for the instruction of introductory programming. The LISP Tutor, (Anderson, 1988) provides instruction on problem solving using LISP, the second and third, BRIDGE (Bonar and Cunningham, 1988) and PROUST (Frye,

Littman, and Soloway, 1988), teach Pascal programming skills, and the last ADA-TUTR (Herro, 1988) teaches Ada.

1. LISP Tutor

The LISP Tutor is actually more of a coach than a tutor. It walks the student through the creation of a LISP function, correcting the student's mistakes whenever he or she deviates from a correct path (Burton, 1988, p. 129). Since the tutor knows what the student is doing at any time, it can use immediate feedback for errors and respond appropriately. The tutor provides instruction and feedback in the context of problem solving. It is based on the principle of presenting the goal structure of the task (Pirolli and Greeno, 1988, p. 193). A menu-driven interface offers the student choices about what goals to attack next, what strategies to use, what code fragment to write down, and so on. The model offers so many choices that any problem solving path the student wants to take is available (VanLehn, 1988, p. 59). These menu choices are used to track the student's cognitive progress and therefore allow the ITS to make a diagnosis. A production scheme is used to represent the goals-and-plans of programming skills and misconceptions (Pirolli and Greeno, 1988, p. 193).

2. BRIDGE

Similar to the LISP Tutor, BRIDGE provides feedback in the context of problem solving. However, unlike the LISP Tutor, BRIDGE does not enforce a top-down refinement of programming goals. Rather, BRIDGE is based on the principle of illustrating how naive goals and plans can be transformed into programming goals and plans (Pirolli and Greeno, 1988, p. 193). BRIDGE was named because it was developed to "bridge" the gap between novice and expert conceptions of programming (Bonar and Cunningham, 1988, p. 410). The

graphically oriented, mouse-driven interface is used in three consecutive phases of program development. The first phase also uses a natural language menu to adequately capture the concepts the student want to express (Figure 1, Bonar and Cunningham, 1988, p. 419).

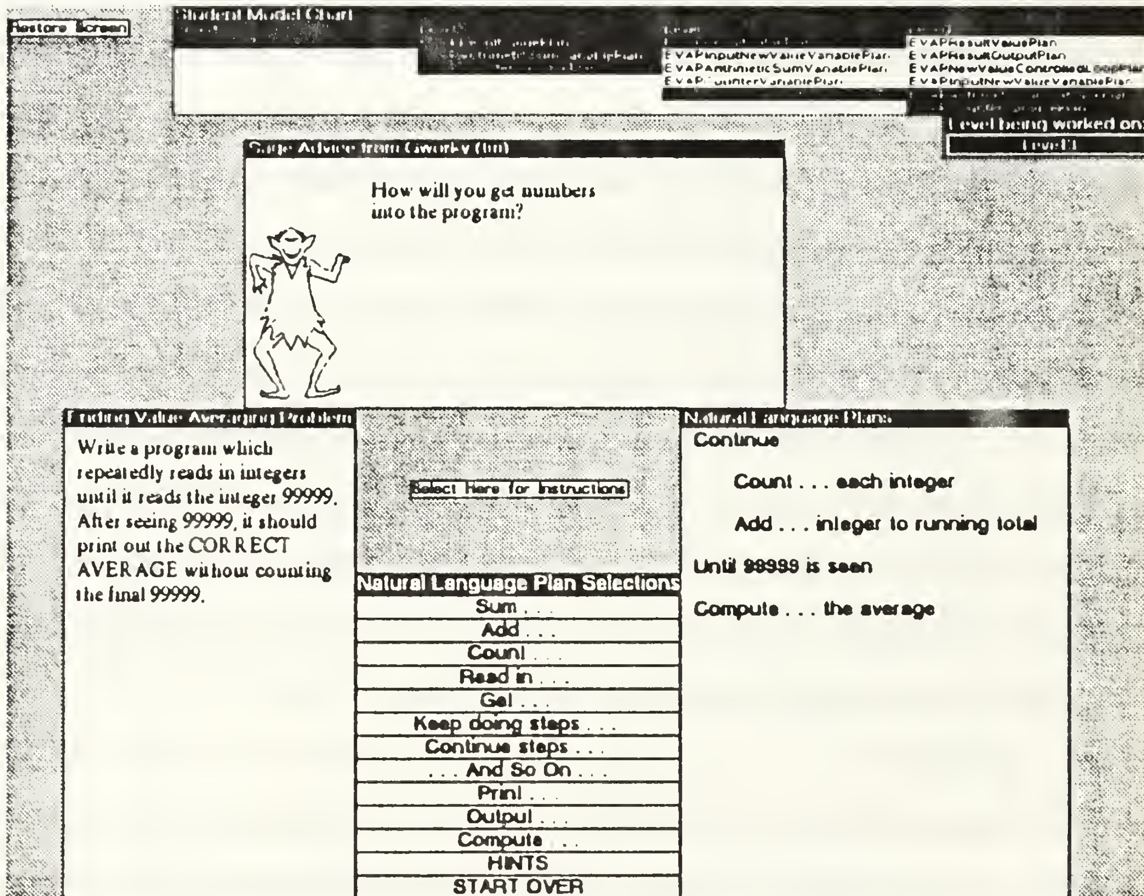


Figure 1 BRIDGE Screen for Phase 1

After a natural language representation of the solution is created, the second phase of development is used to associate the plan with the appropriate pseudocoded tiles (Figure 2,Bonar and Cunningham, 1988, p. 426).

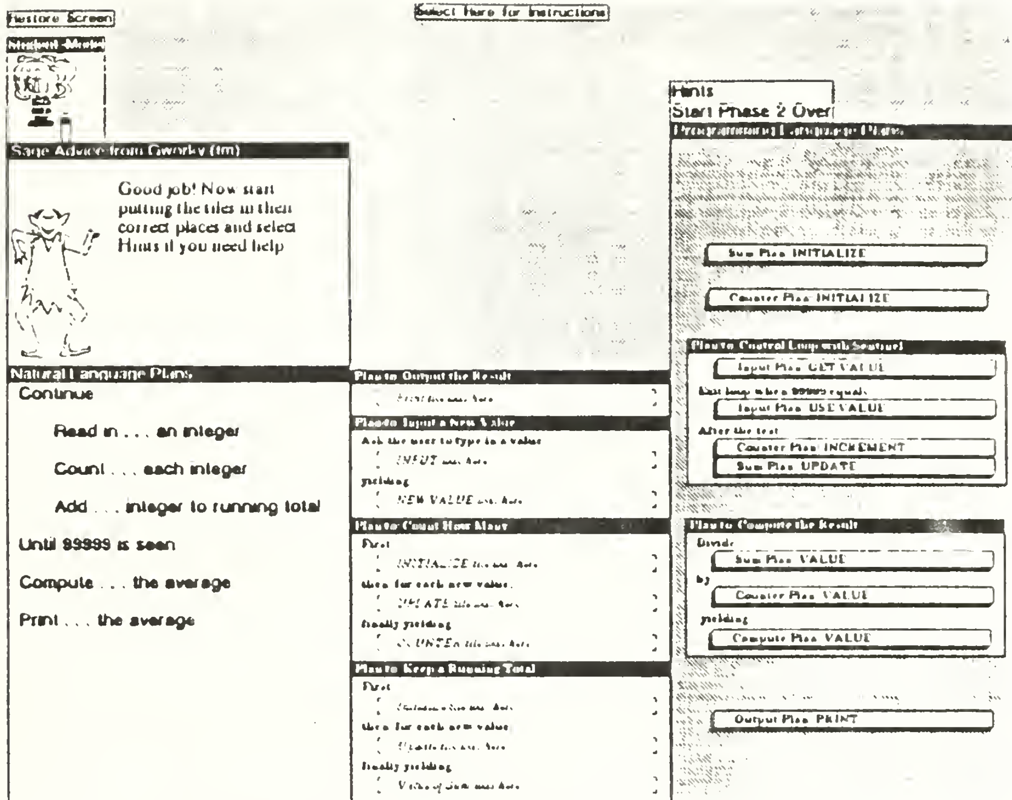


Figure 2 BRIDGE Screen for Phase 2

Finally, during phase 3, the tiles are translated into the corresponding Pascal code fragments (Figure 3, Bonar and Cunningham, 1988, p. 429).

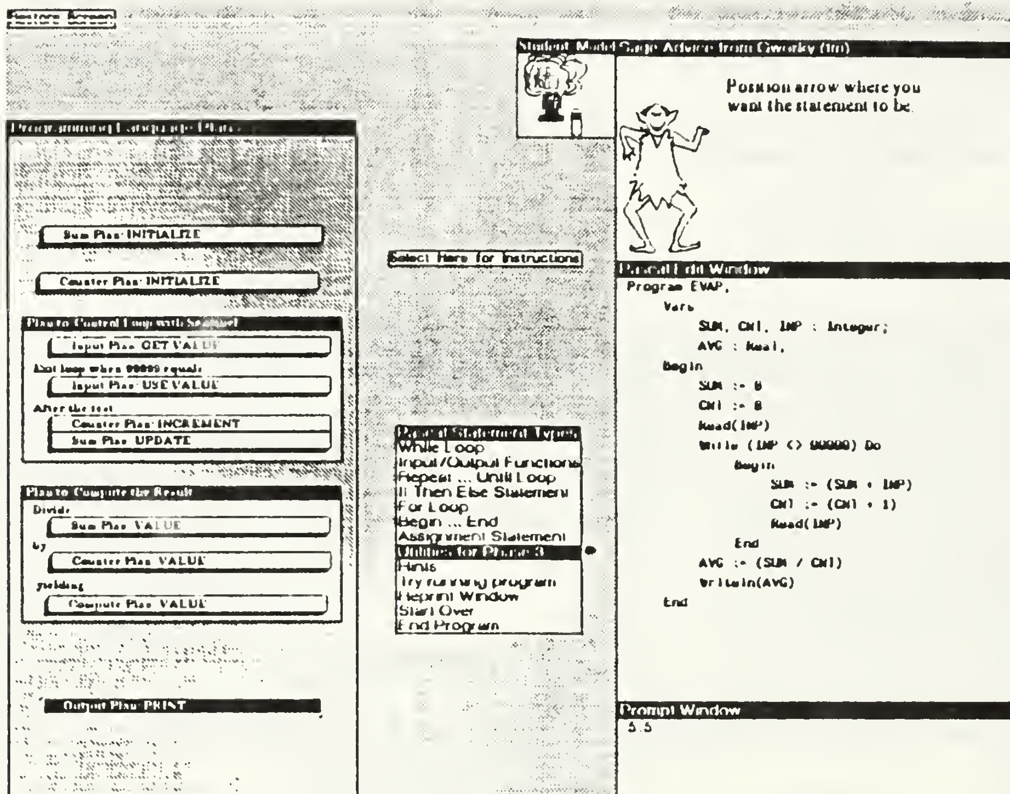


Figure 3 BRIDGE Screen for Phase 3

Since there is not necessarily a simple match between a plan component and Pascal code, students may make a reasonable selection that BRIDGE does not accept. However, in most cases there is a clear correspondence between the initial goal and the final Pascal program.

The knowledge about how the problem should be solved is contained in the problem definition. The requirements for a correct solution to the problem are associated with a problem-specific part of each plan specification. These requirements include information about what phrases should appear in the program and the correct order of those phases. If there are discrepancies between the student's plan and the requirements, an animated instructor, named

Gworky, will appear on the screen to give encouragement or suggest some helpful hints.

3. PROUST

PROUST, a knowledge based program understanding system which does online analysis of completed Pascal programs, is intended to be a tutoring system to assist novice programmers in learning how to program. It consists of two components: a programming expert which can analyze and understand buggy programs, and a pedagogical expert that knows how to effectively interact with and instruct students.

PROUST has a knowledge base of programming plans and strategies, together with common bugs associated with them, all developed around a limited domain aimed at teaching beginning programming. Given a program and nonalgorithmic description of the program requirements, PROUST tries to find the most likely mapping between them by reconstructing the design and implementation steps that the programmer has gone through when writing the program. This reconstruction process is meant to recognize the plans used by the programmer.

Program bugs are characterized as properties of the relationship between programs and intentions. They are directed when there are mismatches between the reconstructed plans and code. Some mismatches are simply recognized, not as bugs, but as common implementation variants of expected code, according to plan transformation rules. Others can be recognized as instances of specific program bugs by bug rules. Other than these two cases, mismatches that cannot be accounted for might result in incomplete or aborted program analysis; or they might be interpreted as bugs, might be ignored, or might bring out warning

messages. Figure 4 shows a sample program recognized by PROUST as a correct solution to a rainfall program. The program requires that the student take in an input stream of rainfall values, and calculates the average, the maximum rainfall on any given day during the period, the number of rainy days, and so forth. Figure 5 shows a buggy solution for the same problem. Figure 6 is the diagnostic output for the buggy program in Figure 5.

```
Program Rainfall(input, output);
   Var Daily Rainfall, TotalRainfall, Maxfall, Average : Real;
     RainyDays, TotalDays : Integer;
  Begin
     RainyDays := 0; TotalDays := 0; MaxRainfall := 0; TotalRainfall := 0;
     Writeln ('Please Enter Amount of Rainfall');
     Readln(DailyRainfall);
     While (DailyRainfall <> 9999) Do
      Begin
        If DailyRainfall >= 0 Then
         Begin
            If Daily Rainfall > 0 Then RainyDays := RainyDays + 1;
             TotalRainfall := TotalRainfall + DailyRainfall;
            If DailyRainfall > MaxRainfall
              Then MaxRainfall := DailyRainfall;
            TotalDays := TotalDays + 1
         End;
         Else Writeln ('Rainfall Must Be Greater Than 0');
       Read(DailyRainfall)
    End;
  If TotalDaysCounter > 0 Then Begin
      Average := TotalRainfall/TotalDays;
      Writeln('Average is: ', Average: 0:2);
      Writeln('Maximum is: ', MaxRainfall: 0:2);
     Writeln('Total Number of Days is: ', TotalDays);
     Writeln('Total Number of Rainy Days is: ', RainyDays)
  End;
  Else Writeln('No Valid Days Entered.');
End.
```

Figure 4 Sample Correct Program for PROUST

```
01  Program Rainfall(input, output);
02  Var Daily Rainfall, TotalRainfall, Maxfall, Average : Real;
03     RainyDays, TotalDays : Integer;
04   Begin
05     RainyDays := 0; TotalDays := 0; MaxRainfall := 0; TotalRainfall := 0;
06     While (DailyRainfall <> 9999) Do
07       Begin
          .
          .
          .
33       End;

  Bug 1: Missing Divide-By-Zero Guard
34      Average := TotalRainfall/TotalDays;
  Bug 2: Missing Output Guard On Average
35      Writeln('Average is: ',  Average:  0:2);
  Bug 3: Missing Output Guard On Maximum
36      Writeln('Maximum is: ', MaxRainfall:  0:2);
          .
          .
          .
        End.
```

Figure 5 Sample Incorrect Program for PROUST

Starting Bug Analysis, please wait ... NOW BEGINNING BUG REPORT:

Now Reporting CRITICAL Bugs in the OUTPUT part of your program:

Bug 1: You need a test to check that at least one valid data point has been input before line 34 is executed. The Average will bomb when there is no input.

Now Reporting MINOR Bugs in the OUTPUT part of your program:

Bug 2: The average is undefined if there is now input. But line 35 outputs it anyway. You should output the average only when there is something to compute the average of.

Bug 3: The maximum is undefined if there is no input. But line 36 outputs it anyway. You should output the maximum only when there is something to compute the maximum of.

BUG REPORT NOW COMPLETE.

Figure 6 PROUST Output for program in Figure 5

4. ADA-TUTR

ADA-TUTR, available through the Ada Repository at White Sands Missile Range in Alamagordo, New Mexico, is an application in computer-assisted instruction. The system is written exclusively for the IBM compatible personal computer. This enables a wide range of screen enhancements, such as inverse video and color. Three homework problems are assigned on the instructional screens and step by step instructions are given in an available textbook. The problems are considered correct if they perform the required calculations on a series of appropriate test values.

C. EVALUATING INTELLIGENT TUTORING SYSTEMS

Intelligent Tutoring Systems can be evaluated based on how well they accomplish four main activities: modeling of knowledge and reasoning, communication, cognitive processing, and tutoring (Woolf, 1988, p. 34).

1. Modeling of Knowledge and Reasoning

A good tutoring system represents the domain knowledge to be taught in such a way that it can reason about that knowledge. Reasoning about the knowledge with the goal to teach is much different than reasoning about the knowledge with the goal to diagnose. If the goal of the system is to teach, it must be able to understand the strategy behind the knowledge representation.

2. Communication

Intelligent Tutoring Systems should take full advantage of the hardware capabilities of the platforms on which they were implemented. If possible, a good tutor will include some combination of simulations, animations, icons, pop-up windows, and pull-down menus (Woolf, 1988, p. 6). Even if such amenities cannot be included, the interface should be intuitive and simple to use. The student's time should be spent learning the domain, not the interface.

3. Cognitive Processing

Cognitive Processing is concerned with modeling a methodology for teaching the domain knowledge, and modeling how a student learns within that domain. In order to effectively teach a student, the system must be able to diagnose whether the student understands the material being presented, and if not, what instructional style and content would enable the student to understand.

4. Tutoring

Once the system interprets the student's understanding of the material, tutoring is employed. Tutoring includes praising, remediation, interrupting, and presenting examples to the student (Woolf, 1988, p. 7). A good tutoring system has the capability to respond to the idiosyncrasies of a student in an effort to motivate the student to continue to use the application.

D. SUMMARY AND EVALUATIONS

The LISP Tutor was developed with the aim of understanding the cognitive modeling process of intelligent tutoring systems and was, at that time (1986), considered revolutionary. However, in the context of all existing tutoring systems, it is quite primitive. The expert module can determine only one possible path for the student to proceed down while solving the assigned problem. This limitation in modeling and reasoning about the knowledge is compounded by the LISP Tutor's lack of communication skills. All descriptions of its interactions (Burton, 1988, p. 129; Pirolli and Greeno, 1988, p. 193; VanLehn, 1988, p. 59) mention the menu-driven system and the clear feedback from the tutor. However, because the system interupts the user whenever he begins down an unavailable path, it is difficult for the student to see the overall changes in logic that occur when even one intended command is not accepted. These constant interrupt will frustrate even the most conscientious student. This style is usually considered coaching, rather than tutoring, because the focus is to perfect the process, that is, get the right answer, even if the student never learns why or how he got it.

The BRIDGE tutoring system is focused on the goal of teaching the student the correct plans needed to solve the problem. Since the progression from plans

to goal is monitored continuously via the phases of development, the student is able to learn as he goes along. In addition, the animated character Gworky is a better means of instructing the student than the cold, harsh mental image of the "computerized instructor." The ability to click and drag the tiles with the mouse enables the user to concentrate on overall logical designs without having to worry too much about the code. By monitoring the selection and placement of the tiles, BRIDGE can interpret the student's thought processes. This is an effective way to learn the computer programming process. However, BRIDGE is implemented for a Xerox machine exclusively.

PROUST receives a complete program produced by the student as input for diagnosis and prints out a comprehensive bug report when the program is diagnosed as incorrect. Its emphasis is on developing the programming expert to understand and debug programs. Relatively less effort has been put in the interactive instructional aspects of the system. PROUST has the ability to model the student's knowledge, as represented in his program, and effectively interpret whether he understands the material. Although PROUST has exceptional diagnostic abilities, the domain is limited to a few basic concepts.

The three systems evaluated above are research products and are limited in their distribution. However, the ADA-TUTR is available through shareware and is easily obtained from the Ada Repository. Although it is advertised as an intelligent tutoring system, it has no facility to model the student's knowledge and, therefore, cannot reason about student progress. There is, in fact, limited communication between the tutor and the student. It is actually an application in computer-assisted instruction with the addition of a few homework problems. Since the only feedback from the system about the homework assignment is

whether it is successful in producing the same results as the intended solution, there is no cognitive processing involved at all. The system cannot determine why one comes up with certain results. It just issues an error message for not having answered the problem properly. The student is then on his/her own to produce a different solution and run the comparison program again.

Our *ITS Ada* contains all major components of intelligent tutoring systems. It instructs the student on the content of the concepts as it evaluates the student's response to the assignment. Therefore, the student is able to assimilate the necessary corrections before being given another problem. The purpose of *ITS Ada* is not only to diagnose any problems that the student might have while learning the concepts, but also to teach the student how to solve the problems. It is able to model the knowledge represented in the student's solution by creating the logical representation embedded in a meaning list. The student's solution is made available to the system by a built-in editor. The interface between the student and the system enables the student to freely navigate the entire domain and receive dynamic instruction. Since the presentation of instructional material depends on the student's performance on previously assigned problems, *ITS Ada* is in fact a tutoring system.

III. *ITS Ada*: ARCHITECTURAL DESIGN

A. COMPONENTS

ITS Ada, an intelligent tutoring system that we have developed, consists of four interrelated components: instructional module, expert module, student module, and diagnostic model, as shown in Figure 7.

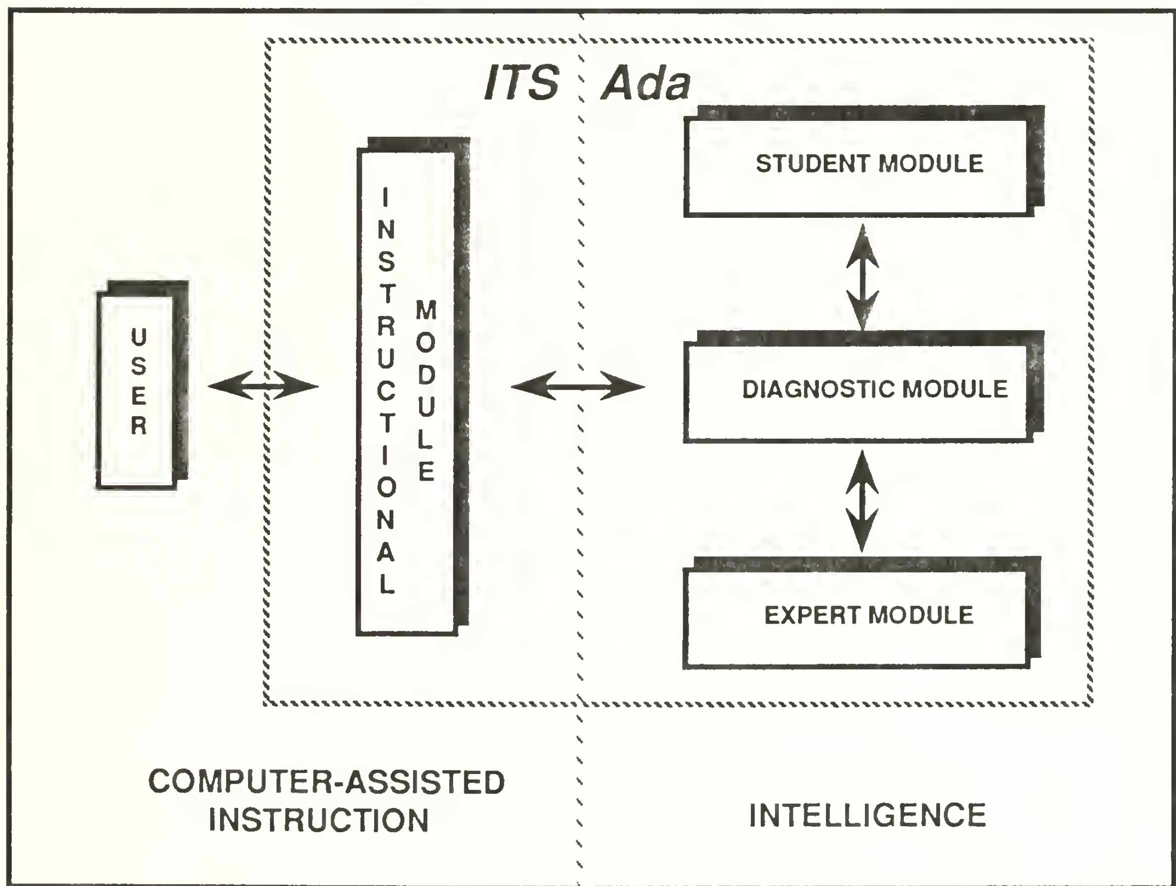


Figure 7 Architecture of *ITS Ada*

An application in computer-assisted instruction uses just the instructional module. This can be viewed as nothing more than an electronic page turner. The instructional screens are presented in a predetermined order. However, when the embedded intelligence of the other three modules is combined, we have an intelligent computer-assisted instruction application. Note that the user is usually not a component of the instructional system. However, it is a critical element in its operation and is essential in evaluating the effectiveness of the system.

B. USER

As with any ITS, there is likely to be extensive variance within the user population. An ITS must be capable of addressing the conflicting characteristics of these students. For example, some learners are impatient with too much review, while others are upset with too little review. A well-designed ITS is able to cater to the rote learner and also permit others to control their progress and learn at their own pace (Johnson, 1988, p. 196).

Although *ITS Ada* makes problems available to help the student apply his newly acquired knowledge, these problems are not necessarily required to follow the intended sequence of presentation. The user may choose either a user specific instructional sequence or a user directed instructional sequence. The former presents assignments and the subsequent instructional sequence is determined by the student's performance on the assignments. The latter option does not present assignments and the student may chose any one of the 12 major topics at the menu and follow through the instructional sequence intended for a student performing well on the assignments. The second option is included so

that the tutor may also be used for reference or review after the student has successfully completed the course.

C. INSTRUCTIONAL MODULE

The instructional module is the interface between the user and the ITS. If this interface is confusing or poorly designed, the effectiveness of the entire session will suffer. Conversely, a well-designed interface can enhance the capabilities of an ITS in many ways. Human interface techniques affect two aspects of ITSs. A well-designed human interface allows an ITS to present instruction and feedback to the student in a clear and direct way. Secondly, the interface determines how the student interacts with the domain. A good interface should ease this interaction (Miller, 1988, p. 143). As a basic guideline, the interface should be kept simple and its use must be intuitive to the student.

ITS Ada was designed to exploit the built-in portability of Ada. It is maintaining this portability that makes the interface a challenge. A highly interactive, mouse-driven interface is not possible because of the close tie between the screen manipulation actions and the machine architecture. Therefore, only crude graphics, simple text input and output, and an elementary editor are used to communicate with the user. For example, the screens may use a series of underscore characters to represent the top and bottom of a box and vertical lines to represent its sides. In addition, only one character is needed to move from screen to screen or between topics. Similarly, the editor uses simple, intuitive commands to allow the student to answer the assigned problems. However, even with such limitations, the screen "lectures" are quite complete and the feedback from the tutor is easily interpreted.

D. EXPERT MODULE

The expert module contains the domain knowledge for the system. The ability to encode and represent the expertise in an ITS is the central focus of developing an expert module. Because of the size of most domains, this is usually the most labor-intensive portion of ITS development. There are three possible options to encode the domain knowledge and the choice of these will, of course, depend on the domain itself. The first option is to try to find some way of reasoning about the domain that does not require our actually codifying the knowledge that underlies human intelligence. This option may require a mathematical model to approximate the underlying domain knowledge. The second option is basically going through the standard stages of developing an expert system. The domain knowledge is usually coded as rules which explain how to perform a given task. The third possibility is going one step further and actually creating a simulation (Anderson, 1988, p. 22).

The knowledge represented in *ITS Ada* covers the complete set of concepts discussed in the Language Reference Manual (LRM) for the Ada Programming Language (DoD, 1983). This information is organized into the following twelve topics: basics of Ada, types, expressions, arrays and strings, input and output, control statements, subprograms, records and dynamic structures, exceptions, files, packages, and tasking. Each of these major topics is presented in a series of 5 - 12 instructional screens and further broken down into subtopics (See Appendix B). These topics and subtopics are organized in an order that is consistent with the topic network in Appendix A. After being introduced to each major topic the student will be given a series of problems associated with that

topic. The solution for each of these problems is contained in a data file, which is accessed during the evaluation of the student's solution to the same problem.

E. STUDENT MODULE

The student module contains the student model. The student model for *ITS Ada* can only track missing conceptions. These are items of knowledge that the student should possess when he or she has mastered the materials taught by *ITS Ada*. Conceptually, the student model is a subset of the expert model. This is called an overlay model because the student model can be envisioned as a piece of paper laid over the expert model. The missing conceptions are holes in the student model which expose those areas that the student is lacking in knowledge (VanLehn, 1988, p. 62).

Conceptually, the student model is a data structure describing the student's knowledge. For this system, the student model is represented by an array which holds the student's identification and the student's degree of mastery in each of the topics monitored. Problems or examples are then displayed, depending on the dynamic level of mastery. After the instructional screens for a topic have been presented, the mastery level is increased. Similarly, after a sequence of problems is successfully completed, the student model is updated appropriately and subsequent actions are performed based on this information. For example, after the presentation of the applicable screens, the student model is updated to reflect the fact that the student has been introduced to that topic. Similarly, after a corresponding problem is assigned, the the student model is updated again to reflect that the student has been exposed to that topic. If two consecutive problems are successfully completed, the student has demonstrated his knowledge

in the topic and the student model is updated to reflect this demonstrated level of mastery.

F. DIAGNOSTIC MODULE

The diagnostic module performs three functions. First, it accepts input from the student for an assigned problem via a simple editor in the instructional module and creates a meaning list, which is a series of records representing either a token or the association of a token to a lexical construct. Second, the diagnostic module checks this solution against the expert's solution and addresses any discrepancies between the two. Third, in response to the interpretation of the student's performance on the assignment, the student model is updated appropriately and subsequent actions are determined. In short, while the student model is the data structure describing the students' knowledge, the diagnostic module manipulates that data (VanLehn, 1988, p. 55).

After the complete sequence of instructional screens have been displayed, a problem definition is displayed and the editor is invoked. The editor prompts the user with an @ sign and provides commands for adding, deleting and modifying lines of program statements. In addition, further modification within each line enables the user to insert and delete individual characters. When the student is satisfied with his response, he may request that his solution be compared to the systems response to the same problem. At this point, the student's response is parsed into a meaning list. However, if a parsing is not possible, the system will inform the student that there is a syntax error and indicate the line where it has been identified. The student will then be able to correct any syntax errors, without penalty, and try again. If the syntax check is successful, a doubly-linked list is created. A new node is added for each token or

lexical construct identified. A lexical construct is recognized when the right hand side components of the embedded rule representing it have also been recognized This process continues until the complete solution has been parsed and its representative meaning list has been created.

Since this meaning list is a logical representation of the student's solution, the student's meaning list can be compared to the expert's list and any discrepancies between the two can be identified. The meaning lists are, in essence, compared from the outside in. The overall structure of the two lists should be identical. For example, if the problem statement asks for the construction of a case statement, both meaning lists will represent case statements. In addition, each component of that overall structure will be compared. Each case statement alternative in the expert's list should also be present in the student's list. Similarly, the sequence of statements associated with each alternative should be present. The diagnostic module is not necessary looking for the two solutions to be identical; but, logically equivalent.

The diagnostic module monitors the overall performance of the student for each topic. Even though several problems may be presented for each topic, it is the overall performance on the set of problems that determines the instructional strategy to employ next. For example, if one student successfully completes the first two problems that have been assigned, they will be treated the same way as another student who gets one right, gets one wrong, and then gets two right in a row. Similarly, if one student gets the first two problems wrong, they will be treated the same way as another student who gets one wrong, gets one right, and then gets two wrong in a row. We believe that the second student will ultimately end up in the appropriate category of performance, i.e., successful or

unsuccessful. However, it may take him longer to prove that he has, in fact, acquired the necessary knowledge. Or, conversely, it may take a little longer for him to prove that he has been unable to assimilate and apply the required knowledge.

G. INTERACTION BETWEEN MODULES

The components of the system interact to ensure that the student receives the proper instruction, in the proper format, at the proper time. The user interacts with the instructional module through the keyboard and the monitor. Simple one character responses are used to allow the student to navigate through the *ITS Ada* domain. When there is an appropriate opportunity for test and evaluation, control is passed to the editor embedded in the diagnostic module.

At the student's request, his proposed solution to an assigned problem is taken from the editor and parsed by the expert module into a meaning list. The parser also parses the expert's solution to the same problem. Both of these meaning lists are then compared in the diagnostic module.

Based on the results of the comparison between the student's and expert's meaning lists, the diagnostic module decides what action to take next. The possibilities are to present another problem, present an example, or transfer back to the instructional module. According to the diagnostic module's evaluation, the instructional module will then present additional screens in a related topic for remedial instruction or move forward through the curriculum and present screens on a superordinate topic. This process continues until the student curriculum is complete or the student decides to quit the session. His next session will begin where he has left off, assuming he uses the same user identification upon resuming the ensuing tutoring session.

Although the parser is the major element of the expert module, the instructional screens can also be considered part of this module since all the knowledge required to successfully complete the assigned problems is contained in some combination of these screens.

IV. *ITS Ada*: IMPLEMENTATION DETAILS

In this section we describe the implementation details of *ITS Ada*. Figure 8 represents the physical files which make up the overall structure of *ITS Ada* and the necessary compilation order.

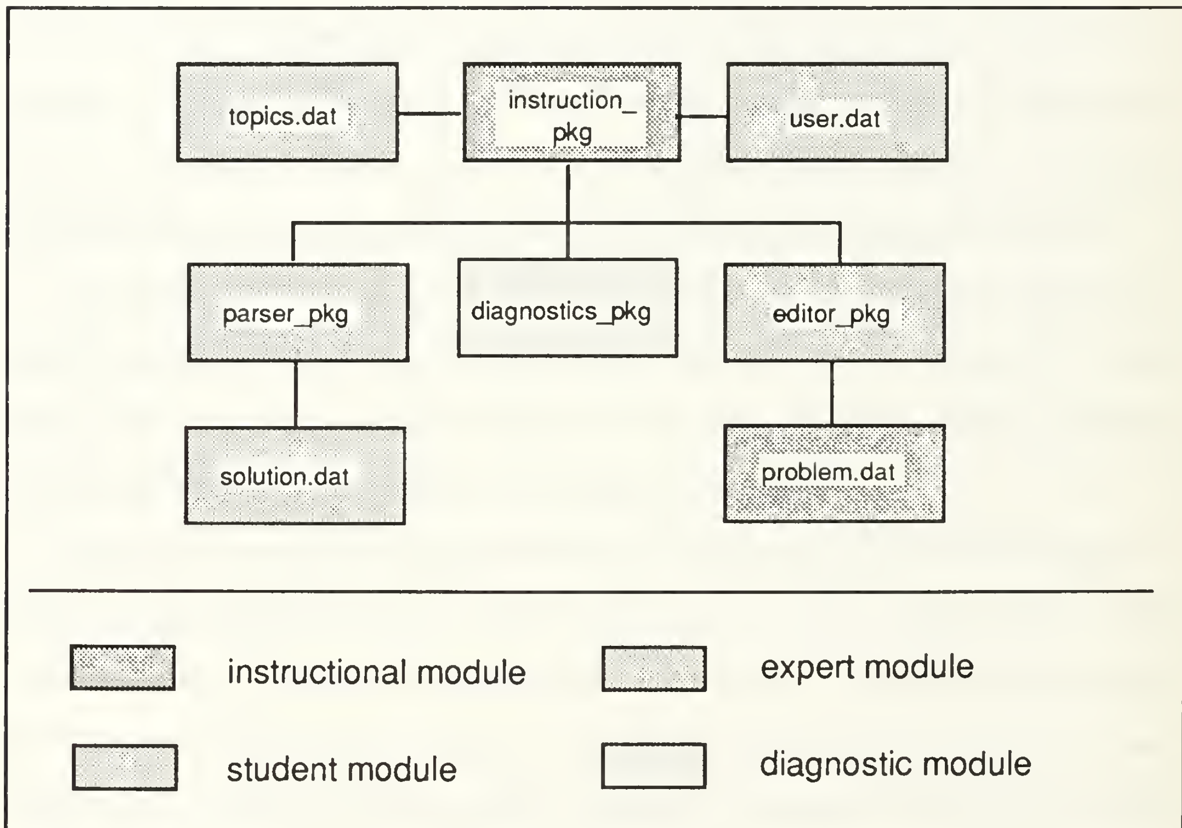


Figure 8 Implementation Diagram for *ITS Ada*

The parser, the editor, and the diagnostics packages must be compiled before the instructional package. In addition, the three supporting files, `topics.dat`,

`problems.dat`, and `solutions.dat`, must be available when the tutor is invoked. The data file `user.dat` will be created when the tutor is used for the first time and will remain on the system thereafter.

A. INSTRUCTIONAL MODULE

1. Instructional Screens

The instructional module presents the lecture portion of the instructional sequence. The screens package accesses the data file `topics.dat` and is used by the screens program. This data file contains 112 screens of text, divided into 12 topics. Each screen is represented by 25 lines of 79 characters each. These proportions were chosen to accommodate a standard computer screen. Each screen is encoded with the chapter and page that it represents. These screens are directly accessible from the screens program.

The screens package contains the procedures used to manipulate the instructional screens. Procedures exist to load screens, display screens and get user information. Each time the load screens procedure is called, 14 screens worth of information are loaded into a buffer. This caching ability minimizes calls to the data file. The screens are loaded at each topic boundary. Therefore, the whole chapter's worth of screens is loaded when reaching the first page if moving forward, or when reaching the last page if moving backward.

In addition, this package contains the procedures used to display the screens. The opening screen, a generic screen, and the menu screen are each treated differently. The opening screen presents 24 lines of text of 79 characters each and a 25th line of 33 characters. The user is then requested to enter his or her user identification. This is necessary to access the data file `user.dat` which contains their student model. Similarly, a generic screen presents 24 lines of text

of 79 characters each and a 25th line of 78 characters. The user is then required to make a choice of which option they would like. Options are available to either move forward to the (N)ext screen, move backward to the (P)revious screen, go back to the (M)enu or (Q)uit the program. If none of these four characters are received, the system will default to moving the user forward to the next screen (Figure 9).

```
                          SCREEN MANIPULATION

When the student is in the instructional sequence, the commands available are

        (N)ext screen            -- next screen or homework problem
        (P)revious screen        -- ignores homework problems
        (M)enu                   -- main menu
        (Q)uit                   -- quits the tutoring system

   While the system is in the user specific instructional sequence, the sequence
of instructional screens will be determined by the user's performance on the
homework problems that have been assigned.  However, when the system is in the
quick reference mode, the sequence of instructional screens is determined by a
topic network and no homework problems will be assigned.  Each topic presented
may be based, at least in part, on material from from previous chapters.
```

Figure 9 Screen Manipulation Instructions for *ITS Ada*

The menu screen (Figure 10) is presented as a generic screen, however, the response requested by the user is an integer value between 0 and 12. At this point, the user has the options of requesting a user specific instructional sequence or a user directed instructional sequence. The former is chosen by using the "0"

option on the menu, while the latter is chosen by using the number that corresponds to the chapter which the user would like to access. The user specific sequence is the normal tutoring mode. This includes the presentation of applicable examples and problems. The user directed option, however, will only present the instructional screens, without any intermittent challenges of problem solving. This option will make this system available for review and reference after the student has completed the course of instruction.

```
                    MENU

*********************************************
*                                           *
*  0  User Specific Instructional Sequence  *
*                                           *
*********************************************

               QUICK REFERENCE

  1  Basics of Ada
  2  Types
  3  Expressions
  4  Arrays and Strings
  5  Input and Output
  6  Control Statements
  7  Subprograms
  8  Records and Dynamic Data Structures
  9  Exceptions
 10  Files
 11  Packages
 12  Tasking
```

Figure 10 Menu Screen from *ITS Ada*

2. Editor

The instructional module includes an editor. Although it is not necessary if the student selects the directed instructional sequence, the editor is an intricate part of the interface for the user specific instructional sequence in *ITS Ada*. When the instructional portion of a topic has been presented, the student will be asked to solve a series of problems. These problems are kept in text form in a data file `problem.dat`. There is a hard-coded table which indicates which lines from the file should be displayed for that problem. After the problem has been printed on the screen, the student will be placed directly into the editor to write Ada code in response to the topic problem just presented. The editing functions available are outlined on an instructional screen (Figure 11).

```
                                  EDITOR

When the student is asked to solve a programming problem, he will be transferred
to the editor.  Although the editor is primitive, it provides all the
capabilities necessary to successfully complete the homework assignments.


(A)dd line                 A 8      -- add lines after line 8
(S)ave                              -- saves as the solution for the problem
(Q)uit                              -- quits the editor
(C)heck                             -- saves and checks the response
(D)elete line #/range      D 8      -- removes line 8 from sequence and renumbers
(P)roblem statement                 -- restates the initial problem
(L)ist                              -- lists all the lines in the file
(L)ist line #/range        L 8-12   -- lists lines 8 through 12, inclusive
(M)odify line #/range      M 8      -- prints out line 8 and enables insertions/
                                       deletions

                           8 The brown fox jumped over the lazy dog.
(I)nsert                       I quick
                           8 The quick brown fox jumped over the lazy dog.
(D)elete                                                       DDDDD
                           8 The quick brown fox jumped over the dog.
```

Figure 11 Editor Instruction for Commands in *ITS Ada*

After editing is complete, the student can then submit the problem to be checked and evaluated by the diagnostic module.

B. STUDENT MODEL

The student model is used to keep track of the knowledge acquired or compiled by the student (VanLehn, 1991). Knowledge acquisition is the process of getting information into one's brain. This can be accomplished by means of reading text and reviewing examples. However, knowledge compilation is the process of cataloging and assimilating this information in one's brain. This can be only be accomplished by engaging in problem solving. Therefore, we have made a distinction between those students who have acquired knowledge and those who have proven that they have compiled the knowledge.

The data file `user.dat`, which is accessed by the screens package is used to build the student model. For this application, the student model is simply an array of records. Each record contains the user identification and their level of mastery for each topic monitored. The student's level of mastery is represented by an enumeration type; *unknown* means that the student has not been introduced to the concepts yet, *exposed* means that the student has been exposed to the information on the instructional screens, and *demonstrated* means that the student has demonstrated that he has acquired knowledge in that area by means of correctly solving a series of problems. In addition, there is also a field to show the last topic which was successfully completed by that student. This file is updated following the completion of each major topic. This is necessary for the user to be able to pick up instruction where he has left off in the previous session.

C. EXPERT MODULE

The expert module contains the domain knowledge for *ITS Ada*. In our case, we need to represent the entire definition of the Ada programming language. Using AYACC (Schmalz, 1988, p. 1), a compiler generator from University of California, Irvine, we produced a parser for Ada. AYACC uses a BNF style specification of a grammar and constructs a state machine which behaves like an expert system . These additional statements are strategically located in those rules that are major Ada program elements. All these rules are identified in the file `ada.y` (Appendix C), which is the input file for the AYACC program. Some of these rules also have an additional statement, {assign_association(yy.rule_id);}, associated with them. This additional Ada statement was added so that a record is added to the growing meaning list when a lexical construct is recognized as being fully parsed. These records, which actually hold the integer value of the rule being parsed are added so that the logic of the overall solution can be maintained and any discrepancies can be identified by name by the diagnostic module.

Since *ITS Ada* was built on the assumption that small representative program constructs, called templates, should be taught to the students, only small problems have been assigned. Ideally, these templates can be combined to create larger blocks of code or complete systems according to the structured programming paradigm. However, the parser produced by the AYACC expects a full compilation unit to be parsed. This obstacle is overcome by associating the problem definition with a particular state in the state machine. For example, if the problem is to create a case statement, the expert module will set up the parser

to expect a case statement. Therefore, the expert module knows both which program construct to expect and when to expect it.

AYACC generates a total of four files: `ada_tokens.a`, `ada_shift_reduce.a`, `ada_goto.a` and `ada.a`. The file `ada_tokens.a` is a package that provides the type and the variable declarations needed by both the parser and the lexical analyzer supplied with the AYACC program. Had this not been available, a complete BNF specification for Ada can be found in the Ada Reference Manual (DoD, 1983, Appendix E).

The files `ada_shift_reduce.a` and `ada_goto.a` are packages used by the parser which contain the tables of states and the progression between states in the state machine. These are generated as separate packages rather than nested within the parsing procedure to prevent them from being pushed onto the stack with each invocation (Schmalz, 1988, p. 2). Although the tables contained in these packages could be placed within a single package, some Ada compilers may have problems compiling the large preinitialized arrays which comprise the tables. The final file, `ada.a`, is the parser itself. It contains the parsing procedure and the additional Ada code that was designed specifically for its function as the expert module for *ITS Ada*.

The parser generated by AYACC expects a full compilation unit as input. Since it was not practical for the solutions to the first problems assigned to require the construction of a complete compilation unit, we tapped into the state machine representation of the parser. The virgin parser initialized the state machine to state 0, thereby setting up the parser to expect a full compilation unit. However, by initializing the parser to the state obtained just prior to parsing the desired program element, the parser will into correctly parse a program

fragment without complaining. The required initialization state is associated with the problem in a preinitialized array contained in the main program. This allows us to maintain the desired modularity of design.

D. DIAGNOSTIC MODULE

As stated before, the diagnostic module performs three functions: accepting a meaning list as input, checking this list for errors, and updating the student model. After the student has completed editing his proposed solution and has submitted the problem to be checked and evaluated, the driver initiates the parsing of that block of code. The parsing is done by the expert module. If there are any syntactical errors in the code, a error message to that effect will be displayed at the point where an error was detected. The student will then be sent back to the editor to make the necessary changes.

On the other hand, if the code has been determined to be syntactically correct, the next phase of evaluation will begin. The parser creates a meaning list which represents the program fragment that has been parsed. This meaning list is a doubly linked list of records. Each record represents either a token or a lexical element. To illustrate, assume the student has been asked to construct a case statement which will associate a particular month with the appropriate season. Both the student's solution (Figure 12) and the expert's solution (Figure 13) will be parsed into a meaning list. Since both the tokens and the lexical constructs are represented in the meaning lists (Figures 14 and 15), any differences between the two meaning lists can be identified by name.

```
1   case MONTH is
2     when DEC | JAN =>
3         SEASON := WINTER;
4     when MAR | APR | MAY =>
5         SEASON := SPRING;
6     when JUN | JUL | AUG =>
7         SEASON := SUMMER;
8     when SEP | OCT | NOV =>
9         SEASON := FALL;
10    end case;
```

Figure 12 Student's Solution

```
1   case MONTH is
2     when DEC | JAN  | FEB =>
3         SEASON := WINTER;
4     when MAR | APR | MAY =>
5         SEASON := SPRING;
6     when JUN | JUL | AUG =>
7         SEASON := SUMMER;
8     when SEP | OCT | NOV =>
9         SEASON := FALL;
10    end case;
```

Figure 13 Expert's Solution

These discrepancies are tracked by a dynamic array which holds data on the token in question and its position in the two meaning lists. If the token appears in both lists, even if it is out of order within the construct, then the token is not considered a problem. However, if the token appears only in the expert's meaning list, it is identified as a missing element. Similarly, if the token appears only in the student's meaning list, it is identified as an unnecessary element. These discrepancies, and the tokens in question, are printed out for the student.

Depending on the student's past performance on problems in this series and the evaluation of the current solution, the student will be given either another problem, an example, or remedial instruction in a related area. The strategy will be determined by the following heuristics outlined in Figure 16.

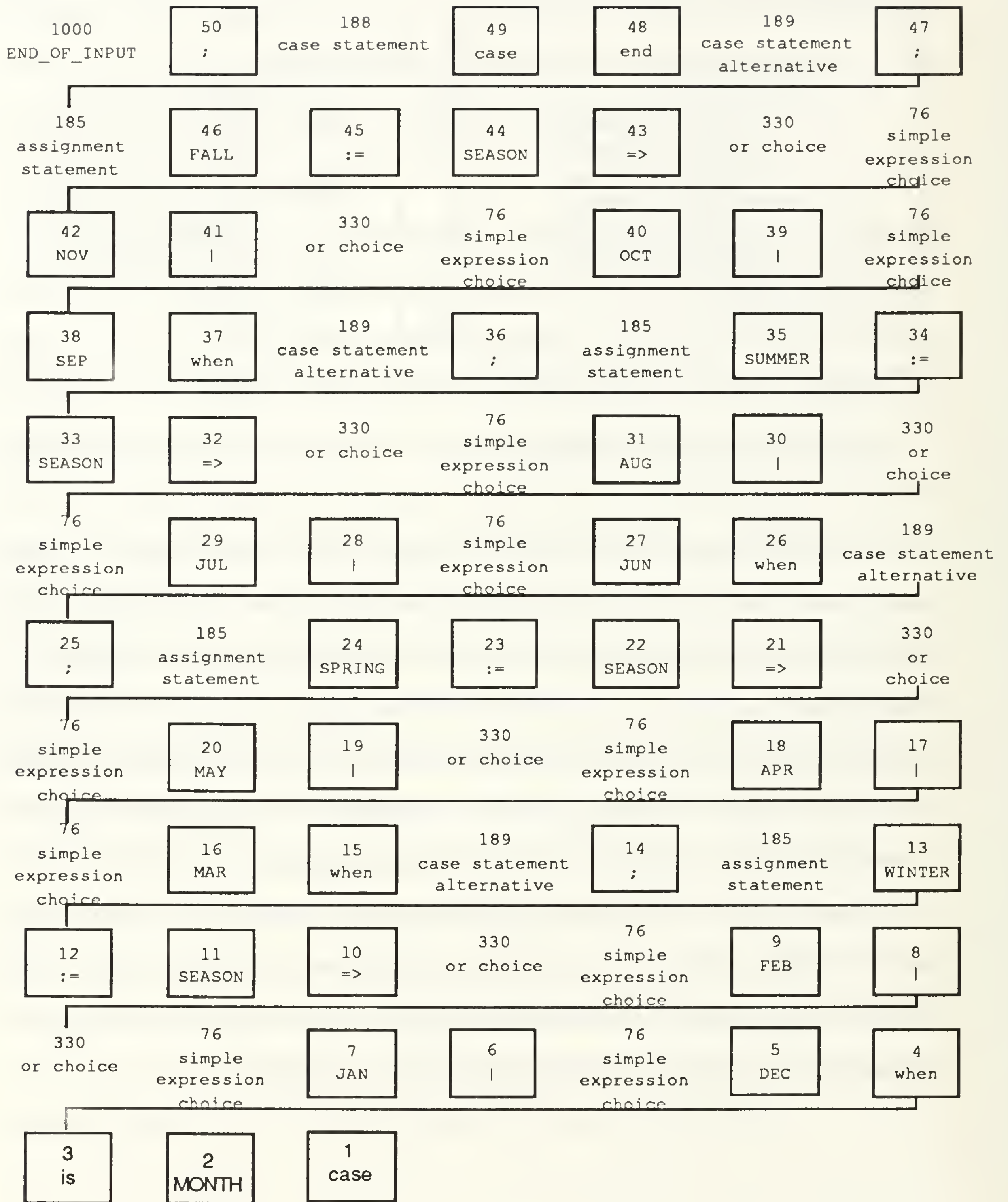


Figure 14 Expert's Meaning List

1000 END_OF_INPUT	50 ;	188 case statement	49 case	48 end	189 case statement alternative	47 ;
185 assignment statement	46 FALL	45 :=	44 SEASON	43 =>	330 or choice	76 simple expression choice
42 NOV	41 |	330 or choice	76 simple expression choice	40 OCT	39 |	76 simple expression choice
38 SEP	37 when	189 case statement alternative	36 ;	185 assignment statement	35 SUMMER	34 :=
33 SEASON	32 =>	330 or choice	76 simple expression choice	31 AUG	30 |	330 or choice
76 simple expression choice	29 JUL	28 |	76 simple expression choice	27 JUN	26 when	189 case statement alternative
25 ;	185 assignment statement	24 SPRING	23 :=	22 SEASON	21 =>	330 or choice
76 simple expression choice	20 MAY	19 |	330 or choice	76 simple expression choice	18 APR	17 |
76 simple expression choice	16 MAR	15 when	189 case statement alternative	14 ;	185 assignment statement	13 WINTER
12 :=	11 SEASON	10 =>	X or [illegible]oice	X 76 si[illegible]e exp[illegible]ion choice	X F[illegible]	X
330 or choice	76 simple expression choice	7 JAN	6 |	76 simple expression choice	5 DEC	4 when
3 is	2 MONTH	1 case				

Figure 15 Student's Meaning List

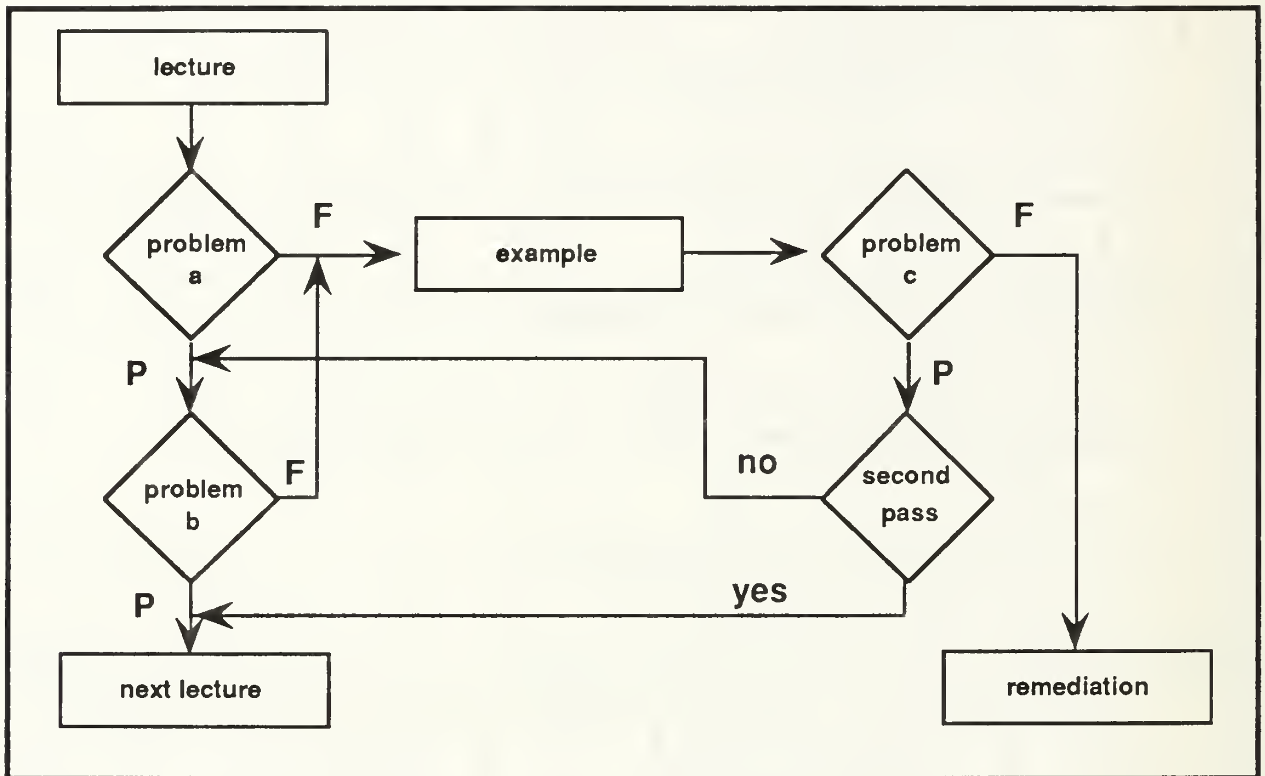


Figure 16 Diagnostic Strategy for Next Presentation

After the lecture portion of the topic has been presented, the first of five possible problems for the topic is assigned. If the first problem, problem a, is completed successfully (shown as P in Figure 16), the second possible problem is assigned. If this problem, problem b, is also completed successfully, then the student model is updated to reflect this exceptional performance. The instruction moves on to another topic. However, if either problem was not successfully completed (shown as F in Figure 16), the next problem data will be presented as an example. An example consists of a problem and a solution. (It is therefore necessary for the diagnostic module to be able to access the solution to an

assigned problem. These solutions are held in text format in the data file `solutions.dat`.) Once the diagnostic module determines that an example should be presented, the problem statement will be displayed, followed directly by the solution. After an example is presented, a third problem, problem c, will always be presented and if two consecutive problems are unsuccessfully completed, the student will be given remedial instruction.

The set of the problem material for this dynamic sequence remains consistent throughout the instruction. However, the format for the problems depends on the previous format and the student's performance. For example, the problem material for the sequence will always will start with problem material 1 in a problem format. However, problem material 2 could be presented as either a problem or an example, depending on the student's performance on the first problem. Similarly, problem material 3 will be presented as a problem, an example, or not at all. Problem material 3 will not be presented if the previous two problems were correctly solved. This cycle continues with the problem material going from 1 to 5, then back to 1 again, and the problem format changing according to the above diagram.

If the diagnostic module determines that remedial instruction is required, the actual topic of that instruction will be consistent with the topic network (Appendix A) constructed for *ITS Ada* based on the layout of the Ada Language Reference Manual (DoD, 1983). This network outlines all subordinate and superordinate concepts that are required for a particular topic. The diagnostic module will determine which of the related subordinate topics should be presented depending on the lexical constructs identified with the problematic discrepancies. These subordinate topics are determined by a case statement in the

diagnostic module. As more or different problems are added to this ITS, this will have to be updated.

E. HARDWARE PLATFORMS

ITS Ada has been tested on three platforms running three different compilers: Zenith 248/InterAda version 4.2.1, Macintosh/Meridian version 4.0.1, and Sun SparcStation/ Verdix Ada version 6.0. It has demonstrated to be completely portable between them. However, modifications had to be made to the basic program to accommodate the Meridian compiler on the Macintosh. Since this Meridian compiler runs in a Macintosh Programming Workshop (MPW) shell, the I/O is line buffered. This means that for same line prompting to work properly, the prompt itself must be stripped out of the input from the screen, leaving only the student's response to be considered.

V. SAMPLE SESSIONS WITH *ITS Ada*

In this chapter, we present two scenarios depicting the interactions between a student and *ITS Ada*. The first scenario is for a good student, one who is able to correctly answer the first two problems presented for a particular topic. The second is for an average student, a user who may have difficulty initially, but eventually solves two consecutive problems, and is therefore able to move on to the next topic.

A. INSTRUCTIONAL MATERIAL

Upon starting a topic in *ITS Ada* all three students will see the same set of instructional screens. All screens for a particular topic are presented before any excersise problems are assigned. For this illustrative example, we will present the sequence of screens for the topic of Control Statements and the problems associated with that topic. This topics includes such subtopics as conditionals, loops, and the case statement. This sequence of screens is shown in Figures 17 through 23.

PARTS OF A PROGRAM 6-1

In simple terms, an Ada program consists of a specification part, a declarative part and a statement part. The specification of a program contains its name and a description of possible parameters to the subprogram. We shall come back to this later. In the declarative part, variables, constants and other parameters can be declared. The part of the program between the 'begin' and 'end' should contain a sequence of one or more statements. Each statement is executed once.

```
subprogram_specification is

   declarative part

begin
   statement 1;
   statement 2;
   .
   .
   .
   statement N;
end subprogram_name;
```

Figure 17 Screen for Parts of a Program

SIMPLE STATEMENTS 6-2

There are several kinds of statements, some are simple statements and some are compound statements. The most common simple statements are assignment statements and procedure calls; we have already seen examples of these. There is also a very simple statement called a null statement. When this statement is executed, nothing happens at all.

Here is an example showing a series of simple statements. The sequence reads in two real numbers and calculates their mean.

```
PUT_LINE("Enter two real numbers");
GET(X1);
GET(X2);
MEAN_VALUE := (X1 + X2) / 2.0;
PUT("The mean is: ");
PUT(MEAN_VALUE);
```

The statement:

```
MEAN_VALUE := (X1 + X2) / 2.0;
```

is an assignment statement and the others are procedure calls.

Figure 18 Screen for Simple Statements

IF STATEMENT 6-3

The most common compound statement is the if statement. The most common way of achieving selection in a program, that is, a choice between two or more different paths in a program, is to use an if statement. An if statement starts with the reserved word 'if' and terminates with the reserved words 'end if.' An if statement is comprised of a 'then' part followed by a number (possibly zero) of 'elsif' parts, ending possibly with an 'else' part.

When the statement is executed, the boolean expression that follow the words 'if' and 'elsif' are evaluated in order from the top down. If any of these boolean expressions are true, the sequence of statements in the corresponding part of the if statement is executed, and then control passes to the first statement after the words 'end if.' If all of the boolean expressions are false, but there is an 'else' part, then that sequence of statements will be executed. If there is no 'else' part, then the if statement terminates without any sequence of statements being executed.

```
if K > 5 or J < 4 then
   K := K + J;
   J := J + 1;
else
   K := J - K;
   K := K + 1;
end if;
```

```
if TEMPERATURE < 50.0 then
   PUT_LINE("Emergency!");
   RAD_SET := RAD_SET + 15.0;
elsif TEMPERATURE < 65.0 then
   PUT_LINE("Too Cold");
   RAD_SET := RAD_SET + 5.0;
elsif TEMPERATURE < 70.0 then
   PUT_LINE("OK");
```

Figure 19 Screen for If Statement

CASE STATEMENT 6-4

We have seen how the if statement can be used to make a selection. A case statement can be used if a choice has to be made between several different alternatives. A case statement starts with the reserved word 'case' and ends with the reserved words 'end case.' After the word 'case', there is a discrete expression whose value determines the choice of one of the several alternatives. A discrete expression is an expression whose value is of a discrete type, that is, the expression is either an integer type or some enumeration type. A list of alternatives following the word 'when' is a list of the possible values that the discrete expression can assume.

```
type ONE_TO_FOUR is range 1..4;
NUMBER            :ONE_TO_FOUR;
...
case NUMBER
   when 1 =>
      PUT("ONE");
   when 2 =>
      PUT("TWO");
   when 3 =>
      PUT("THREE");
   when 4 =>
      PUT("FOUR");
end case;
```

Figure 20 Screen for Case Statement

ALTERNATIVES 6-5

If any possible values are omitted from the list of alternatives, there must be a special 'others' alternative. The 'others' alternative must come last in the case statement, so that when the case statement is executed, the 'others' alternative is reached only if the selector has a value other than those already enumerated in the earlier alternatives. The different alternatives in a list of alternatives can be enumerated with a vertical line or the interval containing them may be stated.

```
type MONTH_NUMBER is range 1..12;
MONTH              :MONTH_NUMBER;
...
case MONTH is
   when 1..2|12 =>
      PUT("WINTER");
   when 3|4|5 =>
      PUT("SPRING");
   when 6..8 =>
      PUT("SUMMER");
   when 9..11 =>
      PUT("AUTUMN");
   when others =>
      PUT("Error in month number");
end case;
```

Figure 21 Screen for Alternatives

LOOP STATEMENT 6-6

To perform iteration in Ada, that is, to execute one or several statements a number of times, a loop statement is used. There are three variations:

(1) a simple loop statement for writing part of a program that is to be executed an infinite number of times

(2) a loop statement with 'while', for writing part of a program that is to be executed a fixed number of times

(3) a loop statement with 'for', for writing part of a program that is to be executed until a certain condition is met

```
loop
   PUT_LINE("HELP! I can't stop");
end loop;

for INDEX in 1..10 loop
   PUT(INDEX);
end loop;

for NUMBER in reverse 1..5 loop
   PUT(NUMBER);
end loop;
```

```
while X > 1.0 loop
   PUT(X,FORE=>6,AFT=>2,EXP=>0);
   X := X / 2.0;
end loop;

while abs(NEXT_TERM) >= EPSILON loop
   SUM := SUM + NEXT_TERM;
   K = K + 1;
   SIGN := -SIGN;
   NEXT_TERM := SIGN / FLOAT(K * K);
end loop;
```

Figure 22 Screen for Loop Statement

```
                         EXIT STATEMENT                      6-7

   There is a special exit statement that can be used in conjunction with the
loop statement.  There are two variants, the first of which is simply:

   exit;

This statement must lie within a loop statement.  When it is executed, the
iteration is terminated and control passes out of the loop statement to the
first statement after 'end loop.'
   The second variant of the exit statement is conditional.  If the boolean
expression is true, then a jump out of the loop statement takes place, just as
in the simple exit described above.  If the boolean statement is not true,
execution continues with the next statement within the loop; no jump takes
place.

   loop                                 loop
      if X > 10 then                       PUT("Enter data");
         exit;                             GET(X);
      else                                 exit when X < 0.0;
         X := X + 2;                       SUM := SUM + X;
      end if;                              PUT(SUM,WIDTH=>3);
   end loop;                            end loop;
```

Figure 23 Screen for Exit Statement

B. SCENARIO ONE

The first problem will be the same for all students. The format will always be interrogatory. If the student correctly solves that problem, the second problem will also be interrogatory and if that one is also answered correctly, the student will move on the next topic. This process is illustrated in Figures 24 and 25. Note that those lines beginning with an @ sign or a number are commands or program statements entered by the student to the editor.

```
Assume the following definitions have been made:

   X, Y, Z                        : INTEGER;

Create a conditional statement that will assign the value of Z to Y
if X < Y, otherwise assign the value of Z to X.

@ a 1
1  if X < Y then
2     Z := Y;
3  else
4      Z := X;
5  end if;
6  .
@ c

1  if X < Y then
2     Z := Y;
3  else
4      Z := X;
5  end if;
<< *** good***>>
```

Figure 24 Good Student: Solved the First Problem

```
Assume the following definition has been made:

   X                              : INTEGER := 0;

Create a loop statement that will increment the value of X by one during each
pass thru the loop and exit when the value of X is equal to 1000.

@ a 1
1  loop
2     X := X + 1;
3     exit when X = 1000;
4  end loop
5  .
@ c

1  loop
2     X := X + 1;
3     exit when X = 1000;
4  end loop
<< *** good***>>
```

Figure 25 Good Student: Solved the Second Problem

C. SCENARIO TWO

The first problem that this student sees is the same as that for the good student. However, because this representative student does not answer the problem correctly, the second problem will be presented in an expository format. Both the problem and the solution will be presented. The student is not expected to answer any questions at this time. He must study the example to learn how to properly solve it. After the student has assimilated this new information, he will move on to another interrogatory problem. When he successfully completes this problem and the next one, he will be able to move on to the next topic. This process is illustrated in Figures 26 through 29.

```
Assume the following definitions have been made:

   X, Y, Z                    : INTEGER;

Create a conditional statement that will assign the value of Z to Y
if X < Y, otherwise assign the value of Z to X.

@ a 1
1  if X < Y then
2     Z := Y;
3  else
4      Z := X;
5  end if;
6  .
@ c

1  if X < Y then
2     Z := Y;
3  else
4      Z := X;
5  end if;
<< *** good***>>
```

Figure 26 Student Solved the First Problem

```
Assume the following definition has been made:

   X                             : INTEGER := 0;

Create a loop statement that will increment the value of X by one during each
pass thru the loop and exit when the value of X is equal to 1000.

@ a 1
1  loop
2     exit when X = 1000;
3  end loop;
4  .
@ c

1  loop
2     exit when X = 1000;
3  end loop;
<< *** missing assignment statement ***>>
    X := X + 1;
```

Figure 27 Student Missed the Second Problem

```
Assume the following definitions have been made:

   Y                             : INTEGER;

   procedure NEGATIVE_ACTION is  . . .
   procedure NON_NEGATIVE_ACTION is . . .
   procedure ERROR is . . .


Create a case statement that will call the procedure NEGATIVE_ACTION when Y is -1
and the procedure NON_NEGATIVE_ACTION when Y is 0 or 1, and the procedure ERROR if Y
has any other value.

@ a 1
1  case Y is
2     when -1 =>
3        NEGATIVE_ACTION;
4     when 0 | 1 =>
5        NON_NEGATIVE_ACTION;
6     when others =>
7        ERROR;
8  end case;
9  .
@ c

1  case Y is
2     when -1 =>
3        NEGATIVE_ACTION;
4     when 0 | 1 =>
5        NON_NEGATIVE_ACTION;
6     when others =>
7        ERROR;
```

```
8  end case;
<< *** good ***>>
```

Figure 28 Student Shown the Third Problem and Solution

```
    Assume the following definitions have been made:

       X,Y                              : INTEGER;

    Create a multiple-branch if statement that will assign the value of -1 to Y if X
is less than 0 and assign the value of 0 to Y if X is equal to 0 or, otherwise, assign
the value of 1 to Y.

    @ a 1
    1  if X < 0 then
    2     Y := -1;
    3  elsif X = 0 then
    4     Y := 0;
    5   else
    6     Y := 1
    7  end if;
    8  .
    @ c

    1  if X < 0 then
    2     Y := -1;
    3  elsif X = 0 then
    4     Y := 0;
    5   else
    6     Y := 1
    7  end if;
    << *** good ***>>
```

Figure 29 Student solved the Fourth Problem

VI. CONCLUSION

A. ACCOMPLISHMENTS

1. A Practical, Fully Functional Ada Tutor

Ada has been designated by the Department of Defence as the mandated language for all new DoD software development. However, there has been great resistance from various sources. One of the major reasons is that Ada is a very sophisticated language and many programmers simply do not want or cannot afford to shift to Ada due to the high cost associated with learning the language. *ITS Ada* has been designed to alleviate this problem by offering a computer based tutor equipped with the many positive qualities of a master teacher. Since the implementation of the initial version has been completed, *ITS Ada* is now a fully functional tutor that teaches complete Ada as defined in the official reference manual (DoD, 1983).

ITS Ada may be incorporated into the computer science curriculum at the Naval Postgraduate School next year. All students graduating from the school are required to demonstrate a practical knowledge of Ada. However, there are numerous students who matriculate with significant experience in a similar computer language. It is far more efficient for these students to spend a few hours with an electronic teacher than to spend many more hours with a human teacher. Their performance can be monitored by an instructor either by administering a conventional final exam or by tapping into the user file which contains all the student profiles. Either way, the student will benefit because the

extra time available can be used to pursue other areas of academic interest. We also plan to make the system available for public use.

2. An Intelligent Tutoring System

ITS Ada contains all major components of a typical ITS and meets all three criteria used by Burns and Capps (Burns and Capps, 1988, p. 1) to evaluate whether a tutoring system is intelligent or not: (1) knowing the domain well enough to draw inferences, (2) able to deduce a learner's approximation of that knowledge, and (3) having strategies to reduce the difference between expert and student performance.

Since the expert module contains the full lexical definition of Ada, the system "knows" the domain well enough to draw inferences. For example, if the key word "case" is presented to the expert module during parsing, an inference is made that a case statement is to follow. The rules embedded in the expert module then determine which additional tokens can be expected to complete a legal parsing of a case statement. Similarly, because the student module maintains a current profile of the student's progress, the system knows the student's level of knowledge in each of the topics monitored and, therefore, of the entire domain. Finally, based on the diagnostic module's interpreted differences, or lack thereof, between the student and expert solutions, the instructional module will either give another problem, show an example, or give remedial instruction on a related topic. The strategy employed will be selected to further reduce the differences between the student and the expert, depending on the student's understanding of the current topic.

3. Implementation in Ada

We implemented the entire *ITS Ada* in Ada, including the AI components which traditionally have been modelled using Lisp or Prolog, to meet the DoD mandate and to make the system available to as many systems as possible. According to the requirements for Ada, all validated compilers must be able to implement a clearly defined set constructs. If our system is implemented using only those defined constructs, the final product will be portable among all machines with valid Ada compilers. In addition to adhering to the ideals of conformance to the Ada standards, we remained within the guidelines of using packages whenever possible. In fact, the system is built from four independent packages. We have tested *ITS Ada* on three platforms, PC, Macintosh, and SunSparcstation, and demonstrated its portability.

B. FUTURE WORK

1. Extended Instructional Materials

ITS Ada uses text descriptions and exercise problems to ensure that the user will be able to develop a working understanding of Ada. A future version may include a greater diversity of instructional materials and exercise problems, interspersed at more frequent intervals so that the user may gain insights into the strengths and limitations of Ada. Since currently individuals are tested only when one entire topic has been presented, a large amount of material is expected to be tested with only a few problems. *ITS Ada* has been designed adhering to the modularity principle. It is easy to incorporate more instructional screens, written for more experienced users, into the system to make it appealing to novice programmers.

At present, structured programming and software engineering principles are not dealt with in *ITS Ada*. However, it is desirable to include them in a future version so that *ITS Ada* is capable of teaching *problem solving* and *program design* in Ada.

The structure of *ITS Ada* can also serve as a model for creating another ITS to teach a different programming language. Although different instructional screens, problems and solutions, and different lexical grammar specifications for the parser are required, the basic control and reasoning mechanism of *ITS Ada* can be adopted. The new development will, therefore, requires considerably less effort than developing a brand new ITS from scratch.

2. Authoring Module

An authoring module can be incorporated into *ITS Ada*'s basic design to allow for more flexibility in the development of problem definition and assignment. The authoring module should ask the domain expert what problems to add, where to add them, and what is his solution to that problem. In order to incorporate problems that are more applicable to the student body, we need to change the problem data file, the array in the driver program, and the required initialization states. No change to the parser is necessary.

3. Timing Constraints When Solving Problems

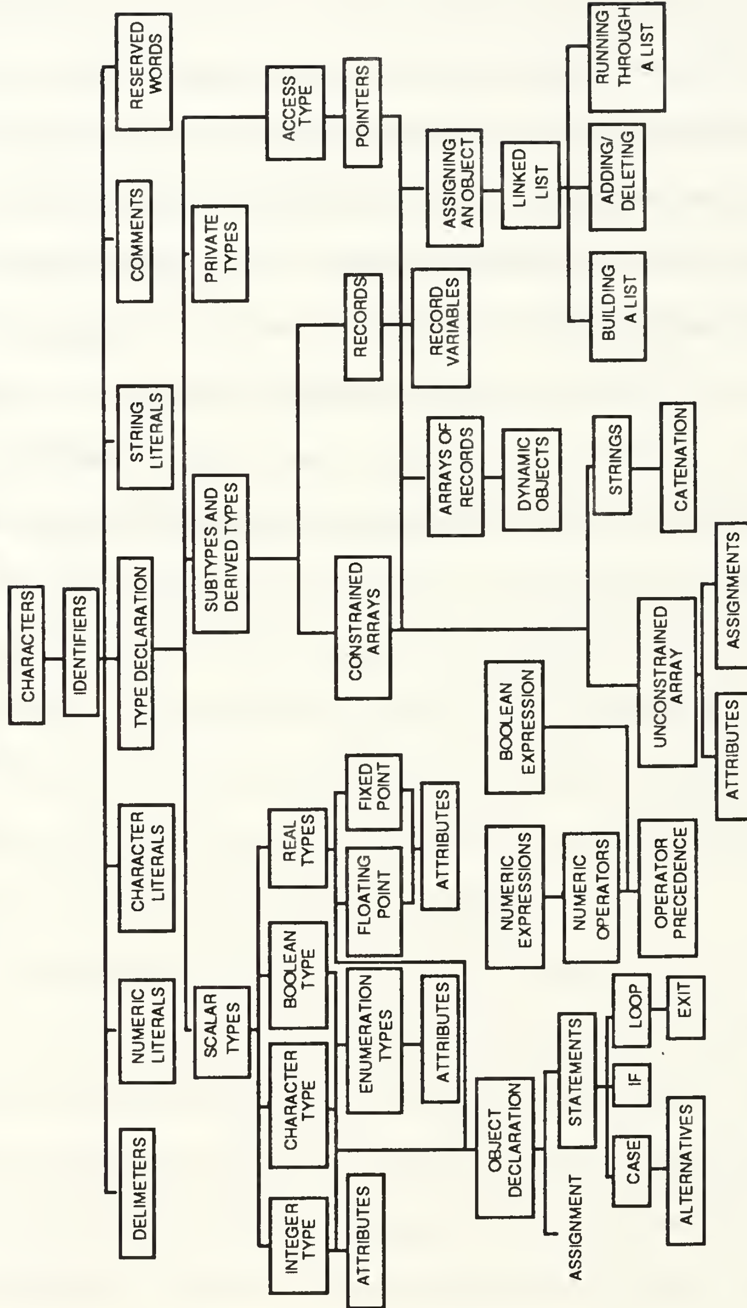
There is a clear correlation between the time needed to answer a problem and the level of knowledge that a student has in that area. Clearly, an intelligent tutor should be able to judge whether a student is spending too much time on certain problems. Tasking is a feature of Ada that provides a timing mechanism in the program. It has not been employed by *ITS Ada*. However, it is essential that a future version of *ITS Ada* has the capability to monitor the

response time for a student to solve assigned problems and evaluate them in a timely manner.

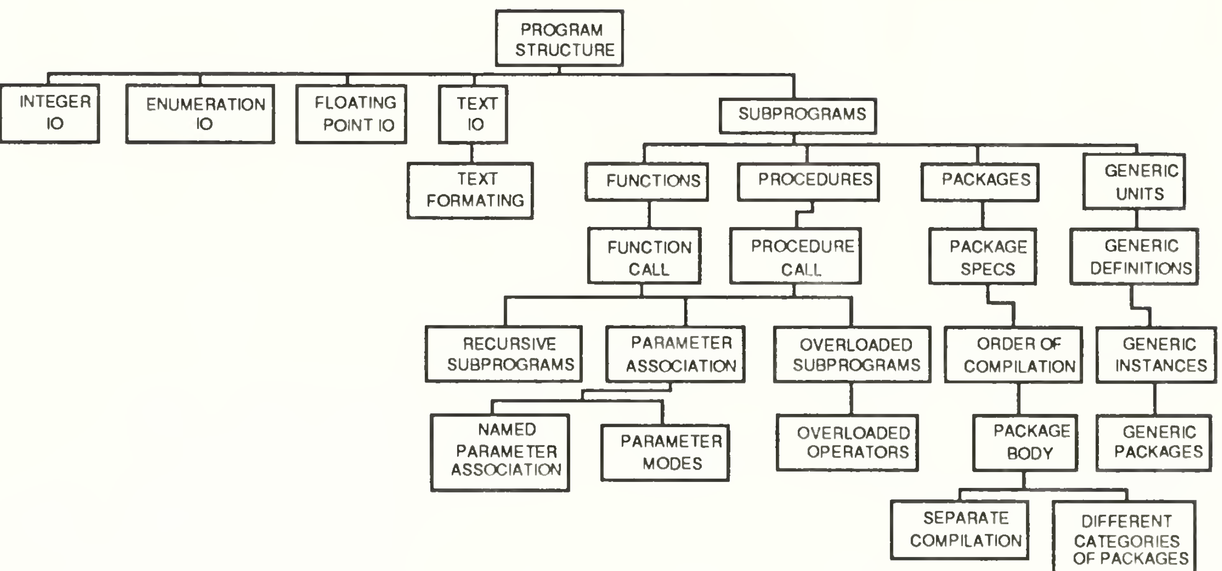
4. Enhanced Instructional Interface

For portability reasons, *ITS Ada* provides only basic user interface support. When a standard windows environment, such as the X Windows, becomes available for more machines, it can become a front-end for *ITS Ada* and be incorporated into the tutor. This would enable us to maintain the high portability goal for *ITS Ada* and offer a more sophisticated and aesthetically appealing interface for the students. Until that happens, we feel that the portability of *ITS Ada* is a more important issue than the enhanced interface. The current interface is adequate and highly intuitive, even the most inexperienced computer programmer can use our system without getting frustrated.

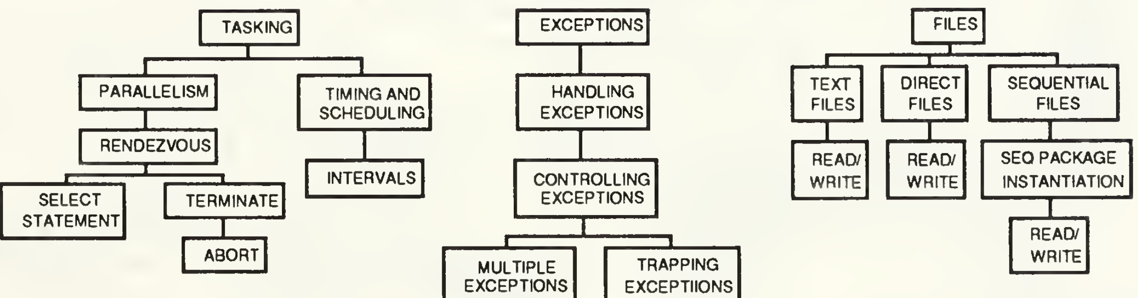
APPENDIX A



TOPIC NETWORK (CON'T)



TOPIC NETWORK (CON'T)



APPENDIX B

SEQUENCE OF SCREENS

TOPIC/SUBTOPIC .... SCREEN

BASICS OF Ada
- SCREEN MANIPULATION 2
- EDITOR 3
- MENU 4
- INTRODUCTION 5
- CHARACTER SET 6
- IDENTIFIERS 7
- DELIMITERS 8
- NUMERIC LITERALS 9
- STRING LITERALS 10
- COMMENTS 11
- RESERVED WORDS 12

TYPES
- DATA TYPES 13
- SCALAR TYPES 14
- INTEGER TYPES 15
- SUBTYPES AND DERIVED TYPES 16
- CHARACTER TYPES 17
- BOOLEAN TYPES 18
- ENUMERATION TYPES 19
- REAL TYPES 20
- ATTRIBUTES 21

EXPRESSIONS
- OBJECT DECLARATION 22
- ASSIGNMENTS 23
- NUMERIC LITERALS 24
- ADDITIONAL NUMERIC OPERATORS 25
- BOOLEAN EXPRESSIONS 26
- OPERATOR PRECEDENCE 27

ARRAYS AND STRINGS
- CONSTRAINED ARRAYS 28
- UNCONSTRAINED ARRAYS 29
- STRINGS 30
- CATENATION 31
- ARRAY ASSIGNMENTS 32

TOPIC/SUBTOPIC	SCREEN
INPUT AND OUTPUT	
PROGRAM STRUCTURE	33
TEXT INPUT/OUTPUT	34
TEXT FORMATING	35
INTEGER OUTPUT	36
INTEGER INPUT	37
FLOATING POINT OUTPUT	38
FLOATING POINT INPUT	39
ENUMERATION INPUT/OUTPUT	40
CONTROL STATEMENTS	
PARTS OF A PROGRAM	41
SIMPLE STATEMENTS	42
IF STATEMENT	43
CASE STATEMENT	44
ALTERNATIVES	45
LOOP STATEMENT	46
EXIT STATEMENT	47
SUBPROGRAMS	
SUBPROGRAMS	48
FUNCTIONS	49
FUNCTION CALLS	51
PROCEDURES	52
PROCEDURE CALLS	53
PARAMETER ASSOCIATION	54
PARAMETER MODES	55
PARAMETER DEMO	56
NAMED PARAMETER ASSOCIATION	57
SCOPE OF A DECLARATION	58
SEPARATE COMPILATION	59
OVERLOADED SUBPROGRAMS	60
OVERLOADED OPERATORS	61
RECURSIVE SUBPROGRAMS	62

TOPIC/SUBTOPIC	SCREEN
RECORDS AND DYNAMIC STRUCTURES	
RECORD TYPES	63
RECORD VARIABLES	64
ASSIGNMENT RECORDS	65
ARRAYS OF RECORDS	66
DYNAMIC OBJECTS	67
ACCESS VARIABLES	68
POINTERS	69
ACCESSING AN OBJECT	70
LINKED LISTS	71
BUILDING UP A LIST	72
ADDING AND DELETING ELEMENTS	73
RUNNING THROUGH A LIST	74
EXCEPTIONS	
EXCEPTIONS	75
DECLARING EXCEPTIONS	76
HANDLING EXCEPTIONS	77
CONTROLLING AN EXCEPTION	78
MULTIPLE EXCEPTIONS	79
TRAPPING AN EXCEPTION	80
FILES	
FILES	81
TEXT FILES	82
READING/WRITING TEXT FILES	83
SEQUENTIAL FILES	84
SEQUENTIAL PACKAGE INSTANTIATION	85
READING/WRITING SEQUENTIAL FILES	86
DIRECT FILES	87
READING/WRITING DIRECT FILES	88

TOPIC/SUBTOPIC SCREEN

PACKAGES
PACKAGES....89
PACKAGE SPECIFICATIONS....90
ORDER OF COMPILATION....91
PACKAGE BODIES....92
RELATIONSHIP BETWEEN SPECS AND BODY....93
DIFFERENT CATEGORIES OF PACKAGES....94
PRIVATE TYPES....96
ARRAY IMPLEMENTATION OF STACK....97
POINTER-BASED IMPLEMENTATION OF STACK....98
GENERIC UNITS....99
GENERIC DEFINITIONS....100
GENERIC INSTANCES....101
GENERIC PACKAGES....102

TASKING
TASKING....103
PARALLELISM....104
RENDEZVOUS....105
SIMPLE SELECT STATEMENTS....106
MORE SELECT STATEMENTS....107
TIMING AND SCHEDULING....108
INTERVALS....109
TERMINATE....110
ABORT....111

```
 ******  **********    ******                   **        *******          **
  ****    ** ***   **   ***  **                ***         **** ***       ***
  ****   **  ***   *   ****    *              ****         ***  ****     ****
  ***        ****      *****                 *****        ****  ****    *****
  ***        ***        ******              ** ***        ****  ****   ** ***
 ****        ***          ******           **   **        ***   ****  **   **
 ***        ****     **   *****           ********        ***  ****  ********
 ***        ****     **    ****          **     **       **** ***   **     **
****        ****     *** ****           **      ***      **** ***   **     ***
******     ******     ******          ****   ******  ********    ****  ******
```

AN INTELLIGENT TUTORING SYSTEM FOR THE ADA PROGRAMMING LANGUAGE

LORI L. DeLOOZE
LIEUTENANT, USN
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA, 93940

WHAT IS YOUR NAME / USER ID:

Screen 1

SCREEN MANIPULATION

When the student is in the instructional sequence, the commands available are

(N)ext screen	-- next screen or homework problem
(P)revious screen	-- ignores homework problems
(M)enu	-- main menu
(Q)uit	-- quits the tutoring system

While the system is in the user specific instructional sequence, the sequence of instructional screens will be determined by the user's performance on the homework problems that have been assigned. However, when the system is in the quick reference mode, the sequence of instructional screens is determined by a topic network and no homework problems will be assigned. Each topic presented may be based, at least in part, on material from from previous chapters.

Screen 2

EDITOR

When the student is asked to solve a programming problem, he will be transferred to the editor. Although the editor is primitive, it provides all the capabilities necessary to successfully complete the homework assignments.

```
(A)dd line               A 8       -- add lines after line 8
(S)ave                             -- saves as the solution for the problem
(Q)uit                             -- quits the editor
(C)heck                            -- saves and checks the response
(D)elete line #/range    D 8       -- removes line 8 from sequence and renumbers
(P)roblem statement                -- restates the initial problem
(L)ist                             -- lists all the lines in the file
(L)ist line #/range      L 8-12    -- lists lines 8 through 12, inclusive
(M)odify line #/range    M 8       -- prints out line 8 and enables insertions/
                                      deletions

                         8 The brown fox jumped over the lazy dog.
(I)nsert                      I quick
                         8 The quick brown fox jumped over the lazy dog.
(D)elete                                                    DDDDD
                         8 The quick brown fox jumped over the dog.
```

Screen 3

```
                          MENU

   ********************************************
   *                                          *
   *  0  User Specific Instructional Sequence *
   *                                          *
   ********************************************

                   QUICK REFERENCE

     1  Basics of Ada
     2  Types
     3  Expressions
     4  Arrays and Strings
     5  Input and Output
     6  Control Statements
     7  Subprograms
     8  Records and Dynamic Data Structures
     9  Exceptions
    10  Files
    11  Packages
    12  Tasking
```

Screen 4

INTRODUCTION 1-1

Most of the text and many of the examples used for this tutoring system are taken from an excellent introductory textbook, ADA FROM THE BEGINNING, written by Jan Skansholm. I would recommend using this book as a supplement to the information presented. Although this course is not intended for the novice programmer, he may find it a useful companion to an introductory course on Ada programming. Experienced programmers will find that this system is an excellent way to learn Ada syntax.

Ada is a modern language with a wide range of uses. It is suitable for both technical and administrative applications. In addition, Ada is a standardized language with strong international support. For most other languages, there are different versions and dialects on different computers; there is only one version of Ada. The name Ada is protected as a brand name. Any implementation of Ada must be validated before it can use the name, i.e. it must go through a battery of tests to check that the Ada standard is adhered to. The precise definition of Ada can be found in a special reference manual. However, it is not intended as a textbook and is therefore rather hard to read, even for an experienced programmer. If Ada is to be used for advanced programming, the manual should be available, but the beginner and 'ordinary' programmer can manage without it.

Screen 5

CHARACTER SET 1-2

The only characters allowed in the text of an Ada program are graphic characters and the format effectors. The characters included in each of the categories of basic characters are defined as follows:

upper case letters A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

lower case letters a b c d e f g h u j k l m n o p q r s t u v w x y z

digits 0 1 2 3 4 5 6 7 8 9

special characters " # & ' ( ) * + , - . / : ; < = > _ |
! $ % ? @ [\ ] ^ ` { } ~

the space character

format effectors horizontal tab, vertical tab, carriage return, line feed,
and form feed

Screen 6

IDENTIFIERS 1-3

Identifiers are used as names for various components of a program and for reserved words. The number of characters permitted in an identifier are limitless and all characters are significant. The first character must be a letter. This may be followed by any combination of numbers, letters or singular inbedded underscores. No spaces are permitted within an identifier since a space is a separator. Lower case letters are interpreted in the same way as the corresponding upper case letter. Therefore, page_count and Page_Count would be identical.

It is good Ada programming style to make identifiers as descriptive as possible. For example, an array of temperatures should be named "TEMPERATURES" rather than "T". In addition, Ada reserved words have special meanings and, therefore, they may not be used as names in a program, These reserved words will be covered in a later section.

```
legal:ID_NUMBER                      illegal:ID-NUMBER    (minus sign)
      PAGE_COUNT                             _POST        (first character _)
      NUMBER_OF_ITEMS                        MILES/HOUR   (/ symbol)
      COLOR                                  1X           (number first)
      ZIP_CODE                               DAY_OF_WEEK_ (ending underscore)
      MILES_PER_HOUR                         PAGE NUMBER  (2 words)
      ANNUAL_PERCENTAGE_RATE
```

Screen 7

DELIMITERS 1-4

Delimiters act to separate syntactic elements (such as operators and variable names) in the same way spaces and carriage returns do. They are used individually:

```
+  -  *  /  (  )  &  '  .  ,  :  ;  <  >  =  |
```

or as two-character compound delimiters. Compound delimiters must stay welded together, with no spaces between the components. The following names are used when referring to Ada's compound delimiters:

=>	arrow	>=	greater than or equal
..	double dot	<=	less than or equal
**	double star	<<	left label bracket
:=	assignment	>>	right label bracket
/=	inequality	<>	box
--	double hyphen		

Screen 8

NUMERIC LITERALS 1-5

There are two classes of numeric literals: real literals and integer literals. A real literal includes a decimal point, while and integer literal does not. The underline character may be used as a delimiter in either case.

A numeric literal may be expressed in the conventional decimal notation (base ten) or in a form that specifies the base explicitly. The base must be between 2 and 16. The form of a based literal is base # number # exponent.

decimal notation

```
12   0      1E6      123_456      -- integer literals
12.0    0.0    0.456    3.14159   -- real literals
1.34E-12       1.0E+6             -- real literals with exponent
```

based notation

```
2#1111_111#     16#FF#     016#0FF#        -- integer literals of value 255
16#F.FF#E+2   2#1.1111_1111_111E11         -- real literals of value 4095.0
```

Screen 9

STRING LITERALS 1-6

A character literal is one character enclosed by single quotes and a string literal is formed by a sequence of graphic characters (possibly none) enclosed in quotation marks.

```
'A'      -- character literal
"Characters such as $, %, and } are allowed in string literals"
```

The length of a string literal is the number of character values in the sequence represented. (Each double quotation is counted as a single character)

```
        ""         -- empty string literal
   " "  "A"  """"  -- three string literals of length 1
```

The quotation marks are not included in the text string, they only act as delimiters. If a quotation mark must be part of the string, it has to be written twice.

```
"He said ""hello"" as he walked into the room."
```

A text string literal must appear on a single line in a program. If it is too long for one line, the the catenation operator can be used.

```
"This is a text string literal that is too long to fit on one" &
"line, so we will write it on two lines and use a special character."
```

Screen 10

COMMENTS 1-7

Comments are used to make the flow of a program clearer and to provide support in writing a program. In fact, it is helpful to create a sketch of the program using comments and then write the code "between the lines."

A comment starts with the compound symbol -- (double hyphen) and extends to the end of the line. A comment can appear at any line of a program. Their presence or absence has an influence on whether a program is legal or illegal, however, their presence does make interpretation easier to the human reader. The reader must be aware that the content of the comment is not necessarily reflected in the code, i.e. the comments may say one thing and the code does something else.

```
X := X + Y;      -- A comment may appear here

Y :=             -- or one may even
     Y - Z;      -- appear here

-- Nothing is stopping you from filling up several
-- lines with comments
```

Screen 11

The identifiers listed below are called reserved words and are reserved for special significance in the Ada programming language. Most Ada programmers adhere to the practice of typing reserved words in lower case letters and all other text in uppercase letters. A reserved word must not be used as a declared identifier.

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

Screen 12

DATA TYPES 2-1

The task of a computer program is to manipulate data objects of various kinds. A data object in a program often represents something that occurs in the real world. Different objects have different properties. In Ada, we say that objects that have different properties have different types. Each object that is used in a program must be declared before it is used and its type stated in the declaration.

A type is characterized by:

(1) the values that can be taken by objects belonging to that type; and

(2) the operations that can be performed on them.

The built-in standard types are INTEGER, FLOAT, CHARACTER, STRING and BOOLEAN. Other types can be defined by the programmer, as needed. Ada is a language that keeps careful check on the types of different objects, that is, objects of a certain type can only take on values that are acceptable for that type.

Screen 13

INTEGER TYPES 2-3

When an INTEGER type is declared, the least and greatest possible integer values that objects of that type can take are stated. The least possible and greatest possible values can be different in different implementations of Ada.

```
        type SAT_SCORE is range 0..1600;
```

where the minimum SAT_SCORE possible is 0 and the maximum SAT_SCORE is 1600.

```
    MAX_LINE                 :constant := 72;
    MAX_COL                  :constant := 17;

    type ELEMENT_LINE is range 1..MAX_LINE;
    type ELEMENT_COL is range 1..MAX_COL;
```

The range of possible values can also be determined by evaluating an expression

```
    type ELEMENT_NUMBER is range 1..(MAX_LINE * MAX_COL);
```

Screen 15

SUBTYPES AND DERIVED TYPES 2-4

A subtype encompasses a limited number of values of a base type. Variables of the subtype and the base type can freely be assigned back and forth.

```
subtype SEVERAL is INTEGER range 5..8;

N            :INTEGER;
M            :SEVERAL;
. . .
N := M;
```

Ada has another mechanism for creating data types out of existing data types. Declarations of derived types look like other type declarations except that they contain the keyword 'new.' The central difference between subtypes and derived types is that a subtype may be mixed with variables of the base type, whereas a derived type must be kept separate from the parent type.

```
type NEW_INTEGER is new INTEGER range 1..100;

P             :INTEGER;
Q             :NEW_INTEGER;
. . .
P := Q;    -- illegal
```

Screen 16

The CHARACTER type is used for handling only single characters, such as letters, digits, special symbols, or non-printing control characters. Non-printing control characters can be used when you want a terminal to do things, for example, begin on a new line, clear the screen or make a bell ring.

In Ada, the type CHARACTER is defined so that it conforms to the ASCII standard. This means that a variable of type CHARACTER can contain any of the 128 characters in the ASCII standard. Each of the 95 graphic characters of this character set is denoted by the corresponding character literal.

For example:

```
type ROMAN_DIGIT is ('I', 'V', 'X', 'L', 'C', 'D', 'M');

type HEXADECIMAL is ('0','1','2','3','4','5','6','7','8','9','A','B','C',
                     'D', 'E','F');
```

The type CHARACTER is an enumeration type. In an enumeration type, there is a relative ordering defined between the different values. The order is decided by how the values are listed in the definition; of two values, the one listed first is considered the lesser. In our definition of ROMAN_DIGITS, 'X' is less than 'C', which is contrary to normal alphabetical order.

Screen 17

BOOLEAN TYPES 2-6

The BOOLEAN type is actually an enumeration type, like the type CHARACTER. However, it is defined with only two possible values, FALSE and TRUE. Note that FALSE < TRUE, but this is not normally significant.

BOOLEAN types are useful as flags; it is good programming style to give them identifiers which describe conditions, for example:

```
LIGHT_ON            :BOOLEAN;
```

The boolean variable LIGHT_ON can have the value 'on' (true) or 'off' (false) with different ambient conditions depending on the value. Therefore, it can be used in a clearly described test for a conditional branch.

```
LIGHT_ON            :BOOLEAN;
. . .

LIGHT_ON := TRUE;
. . .

if LIGHT_ON then
   CHECK_WATER_LEVEL;
end if;
```

Screen 18

REAL TYPES 2-8

Real types are the non-discrete scalar types that are used to represent numbers. In Ada, there are two categories of real types, namely floating point and fixed point types. The difference between these two types has to do with the nature of real numbers and how they are represented in computer systems.

A floating point number can represent a very large value or a very small number. When we create a floating point type, we can specify the number of decimal digits of precision the compiler needs to keep track of and the range of acceptable values. FLOAT is the floating point type available in all implementations of Ada. Other derived types can also be defined.

```
type PRECISE_MEASUREMENT is digits 15;
type PERCENTAGE is digits 4 range 0.0 .. 100.0;
```

A fixed point looks alot like the floating point type, however, instead of using the keyword 'digits,' the fixed point type uses the keyword 'delta.' This is called an accuracy constraint and tells how many digits will be used to represent each number. Any number of either real type must have a decimal.

```
type DOLLAR is delta 0.01 range 0.00 .. 10000.00;
type FINED_TUNED is delta 0.000025;
type THOUSANDTHS is delta 0.001;
```

Screen 20

OBJECT DECLARATION 3-1

An object declaration declares an object whose type is given. The declared object is a constant if the reserved word 'constant' appears in the object declaration; the declaration must then include an explicit initialization.

```
HIGH_LIMIT          :constant INTEGER := 10000;
LOW_LIMIT           :constant INTEGER := 0;
```

An object that is not a constant is called a variable. The only ways to change the value of a variable are either directly by an assignment, or indirectly when the variable is updated by a procedure call.

```
COUNT               :INTEGER;
SIZE                :INTEGER range 0 .. 1000 := 0; -- initial value is 0
SORTED              :BOOLEAN := FALSE; -- initial value is FALSE
```

A number declaration is a special form of constant declaration. The constant declared by a number declaration is called a named number and has the type of the static expression.

```
PI                  :constant := 3.14159;  -- a real number
MAX                 :constant := 500;      -- an integer
```

Screen 22

ASSIGNMENTS 3-2

The compound symbol := is called the assignment symbol and is used to denote assignment. Assignment means that whatever is on the right hand side of the assignment symbol is placed in the variable on the left hand side. The variable must be of the same type as whatever is on the right hand side of the symbol. Any variables that may appear on the right hand side are not affected by the assignment. Their values remain unchanged.

Assignment can be made when objects are declared:

```
SALES_TAX                  :constant := 0.065;
NUMBER_OF_ITEMS            :INTEGER range 0 .. 1000 := 0;
OPEN                       :BOOLEAN := TRUE;
```

Or assignments can be made explicitly by using an assignment statement:

```
K := INDEX + 15;
X1 := 23.5;
ALARM := TEMP > 200.0;
NAME := "JONES";
```

When an assignment statement is read, the := symbol should be interpreted as "becomes" or "gets". For example, K "becomes" the value INDEX + 15.

Screen 23

ADDITIONAL NUMERIC OPERATORS 3-4

In addition to the basic operators of addition, subtraction, multiplication, and division, Ada provides the remainder (rem) and modulus (mod) operators. The rem operator can be used to find the remainder after division. This needs two integer parameters of the same type. The expression 12 rem 5, for example, gives the result 2, the remainder when 12 is divided by 5. By definition, A rem B has the same sign as A and an absolute value less than the absolute value of B. The operator mod works in almost the same way. A mod B has the same sign as B and an absolute value less than B.

A	B	A rem B	A mod B
12	5	2	2
12	-5	2	-3
-12	5	-2	3
-12	-5	-2	-2

There is also an exponential operator denoted by **. The result will be the same type as the first operand. The second operator can only be an integer. N ** 5 is identical to N * N * N * N * N and X ** (-4) = 1/(X * X * X * X).

The final numeric operator is the absolute value (abs). The absolute value of a positive operand is itself and the negated operand if it is negative.

abs(10.3) = 10.3 abs(-50) = 50

Screen 25

BOOLEAN EXPRESSIONS 3-5

It is also possible to build an expression whose value has type BOOLEAN. Such an expression is called a boolean expression. Boolean expressions use either relational or boolean operators. Relational operators can be used to make comparisons, such as equality or inequality. The two operands of a relational operator must be of the same type.

=	-- equal to	<=	-- less than or equal to
/=	-- not equal to	>	-- greater than
<	-- less than	>=	-- greater than or equal to

Both the operands and the result of boolean operators are of the type BOOLEAN. These operators are NOT, AND, OR, XOR, AND THEN and OR ELSE. The NOT operator changes a TRUE to a FALSE and vice versa. The operators AND and OR have their natural logical meanings. The operators AND THEN and OR ELSE are exactly the same as AND and OR, respectively. The difference is that the left operand is evaluated first. The right operand is then evaluated only if it is necessary.

A	B	NOT A	A AND B	A OR B	A XOR B
TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE

Screen 26

OPERATOR PRECEDENCE 3-6

When complicated expressions are constructed, it is important to know the order in which the component expressions will be evaluated.

```
**    abs    not

*      /     mod    rem

+      -

=     /=      <     <=      >      >=

and    or    xor    and then    or else
```

Operators on the same line have the same precedence. The top line of operators has the highest precedence and the bottom line of operators has the lowest precedence.

```
5 ** 6 > 5 + 20 and 25 / 5 = 6 rem 4
  30   > 5 + 20 and 25 / 5 = 6 rem 4
  30   > 5 + 20 and    5   =    2
  30   >   25   and    5   =    2
      TRUE         and      FALSE
                 FALSE
```

Screen 27

CONSTRAINED ARRAYS 4-1

The scalar types we have declared so far have been simple types where each object of that type assumes only a single value. In an array type, however, an object consists of a numbered collection of similar components. It can also be said that an object of an array type is a kind of table in which each element has a particular number associated with it.

We shall start by looking at constrained array types. When a constrained array type is declared, both the numbering of the components and the types of the individual components must be specified, that is the table has been drawn and we know the format of the values to be entered into the table.

```
1  2  3  4  5  6  7  8   -- the eight hours of the work day
6  3  2  0  4  9  4  6
```

This is represented by: `type COUNTS is array(1..8) of INTEGER;`

Similarly, if we wanted to keep counts for a whole week, we could build a 2 dimensional array, with 8 columns (hour) and 7 rows (day of the week).

```
type DAY_OF_WEEK is
     (MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY,SUNDAY);
type HOUR_OF_DAY is range 0..8;

type COUNT_ARRAY IS (DAY_OF_WEEK,HOUR_OF_DAY) of INTEGER;
```

Screen 28

When the components of one-dimensional array are characters, the array is called a STRING. The STRING type is built into each implementation of Ada. The STRING type is defined as follows:

```
type STRING is array(POSITIVE range <>) of character;
```

A text string variable may appear as follows:

```
NAME          :STRING(1..10) := "JOHN SMITH";
```

In this case, the object NAME will hold 10 characters, numbered from 1 to 10. We can select a particular element of the STRING using indexing.

NAME(7) refers to the seventh character of the string, ie "M".

A similar construct can be used to create a slice (part of a string).

NAME(2..4) refers to the second to forth characters, inclusive, ie "OHN".

The string can be changed by using slicing.

```
NAME(1..5) := "MARK ";        NAME now has the value "MARK SMITH".
```

Screen 30

CATENATION 4-4

It is possible to join strings together in sequence, to catenate strings, using the operator denoted by the symbol &. It is possible for 1 or both strings to have type CHARACTER

```
    NAME := "Tom" & "my";            -- result is "Tommy"
    NAME := "Ji" & NAME(3..5);       -- result is "Jimmy"
    NAME := NAME(1..4) & 'i';        -- result is "Jimmi"
```

Just as for text strings, the operator & can be used for catenating arrays.

```
    type VECTOR is array(INTEGER range <>) of FLOAT;

    VI              :VECTOR(1..2);
    V2              :VECTOR(101..103);
    V3              :VECTOR(0..4);

    V3 := V1 & V2;
or  V2 := 27.0 & V1;
```

Note that the array sizes must be compatible, ie a 2 member array and a 3 member array can catenate and be copied into an array with 5 or more members.

Screen 31

Using aggregates, array variables can be initialized when they are declared.

```
type MONTH_TYPE is (JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC);
type NO_OF_DAYS is array(MONTH_TYPE) of INTEGER;

DAYS_IN_MONTH          :NO_OF_DAYS := (31,28,31,30,31,30,31,31,30,31,30,31);
```

When there is a large array where several components are to be given the same value, the reserved word others can be used.

```
type VECTOR is array(1..10) of FLOAT;
VECTOR_A               :VECTOR := (1.0,2.0,3.0,others => 0.0);
VECTOR_B               :VECTOR := (others => 0.0);

VECTOR_A := (others => 0.0);
NAME := (others => ' ');
```

When assigning values to an array, it is necessary for the array on the right hand side to have the same type and same number of components as the array on the left hand side.

```
VECTOR_A := VECTOR_B;
```

Screen 32

PROGRAM STRUCTURE 5-1

```
with ...;      -- with GEOMETRY;

use ...;       -- use GEOMETRY;

procedure PROGRAM_NAME is

   declarations

begin

   series of statements

end PROGRAM_NAME;
```

Ada allows you to write your program in small separate sections and to link these sections together after they have been compiled. If one section needs access to another section, the "with" clause is used. Items declared in this other section can be accessed using selection or dot notation. For example,

```
FIELD_AREA            :GEOMETRY.AREA;
```

refers to the type AREA in the package GEOMETRY. However, dot notation is not necessary if the use clause is introduced.

Screen 33

TEXT INPUT/OUTPUT 5-2

In all implementations of Ada, we have access to the resources of a standardized package called TEXT_IO. It will enable us to read and write values of the types STRING and CHARACTER. The package contains several tools, including PUT, which will allow us to write text at a terminal. We can do this by writing TEXT_IO.PUT in the program or by "using" TEXT_IO and writing just PUT. It is good Ada programming style to use dot notation for everything except TEXT_IO. A PUT_LINE works the same way as a PUT, but a new line is started after the output.

```
PUT(TEXT);        -- TEXT can be either a character or a string
PUT_LINE("This is the line to be printed");
```

In order to read characters and strings, two different versions of GET are used. A GET with an object of type CHARACTER will take in the next character to be read in and associate it with that object. Line or page terminators will be skipped.

```
GET(CHAR);     -- CHAR is type CHARACTER
```

Similarly, a GET with an object of type STRING will read in characters one by one placing them into the array starting on the left. Line and page terminators will be skipped. The string must be COMPLETELY FILLED.

```
GET(LAST_NAME);    -- a 10 character string will require 10 characters
```

Screen 34

To read a whole line in at once, or to get a string without having to pad it with spaces at the terminal, there is a special procedure called GET_LINE. GET_LINE will get the value of the string and its length. Reading normally ends when a line terminator (carriage return) is met in the input stream. Reading can also end if the object is not as long as the string that is read in. If that object to which nothing has been read will be undefined. This is why it may be a good idea to initialize any object of type STRING with spaces.

```
GET_LINE(NAME,NAME_LENGTH);
```

TEXT_IO provides a way of positioning both input and output by using NEW_LINE and SKIP_LINE. NEW_LINE causes a line terminator to be output. If we supply a spacing parameter, we can get multiple line terminators output.

```
NEW_LINE;
NEW_LINE(5);
```

For positioning input, usually to skip over any garbage left on the input line by the user, we have SKIP_LINE. SKIP_LINE scans over the input stream until a line or page terminator is detected. A spacing parameter may also be used.

```
SKIP_LINE;
```

Screen 35

INTEGER OUTPUT 5-4

The versions of PUT and GET or writing and reading characters and text are always present in TEXT_IO. However, this is not the case for numeric and enumeration types. Since it is possible to work with many different integer, float and enumeration types, it is not possible for TEXT_IO to contain a version of PUT for every type imaginable. Instead, it has templates for input and output of integer types, floating point types and enumeration types. For example, to be able to read and write integers, the following (or similar) must be placed among the declarations in a program.

```
package INTEGER_INOUT is new INTEGER_IO;
use INTEGER_INOUT;
```

When writing integers to the terminal, the number of characters to be printed can be specified or assumed to be the default value, If the number to be printed is smaller than the width specified, padding takes place with blanks to the left of the number and if the number to be printed is larger than the width specified, the exact number of position required are allowed.

```
NUMBER                  :INTEGER := 54321;

PUT(NUMBER);                   ^^^54321
PUT(NUMBER,WIDTH=>5);          54321
PUT(NUMBER,WIDTH=>1);          54321
```

Screen 36

INTEGER INPUT 5-5

To read in integers, a special input/output package must be created using the package INTEGER_IO in TEXT_IO, just as was done for output.

```
type WHOLE_NUMBER is range -1000..1000;

package WHOLE_NUMBER_INOUT is new INTEGER_IO(WHOLE_NUMBER);
use WHOLE_NUMBER_INOUT;
...
GET(W);
```

The procedure GET expects something of the type WHOLE_NUMBER. In the input, this is given as a series of characters, for example, the integer 475 is represented by the three characters '4', '7', and '5'. Input continues for as long as the characters read can be interpreted as part of an integer. If the integer read is not compatible with that of the type expected, an error will result.

The WIDTH parameter can be used if data are to read that have been written in a special way, with a particular number of digits in the input stream.

```
GET(W,WIDTH=>4);
```

For example, -157890 written at keyboard becomes -157.

Screen 37

FLOATING POINT OUTPUT 5-6

For floating point numbers, as for integers, an individual package must be created for input and output, this time using the package FLOAT_IO in TEXT_IO.

```
package FLOAT_INOUT is new FLOAT_IO(FLOAT);
use FLOAT_INOUT;
```

In FLOAT_IO, there is a version of PUT. Parameters for the number of digits before the decimal place (FORE), the number of digits after the decimal place (AFT) and the number of digits in the exponent (EXP) can be specified.

```
P := -123.4;
Q := 0.00567;

PUT(P);                              -1.23400000E+02
PUT(Q);                               5.67000000E-03
PUT(P,AFT=>2);                       -1.23E+02
PUT(Q,FORE=>5,AFT=>1);                5.7E-03
PUT(P,EXP=>4);                       -1.23400000E+002
PUT(Q,EXP=>0);                        0.00567000
```

Screen 38

To read values into variables of floating point types, the package FLOAT_IO in TEXT_IO must be used to create an input/output package for the particular floating point type in question.

```
type TEMPERATURE is digits 4;

package TEMPERATURE_INOUT is new FLOAT_IO(TEMPERATURE);

T                    :TEMPERATURE;
...
GET(T);
```

Input works in the same way as for integers, except the format rules for input of a real number differ from those for an integer. Data can be input either in ordinary form, with figures before and after the decimal point, or in exponent form. A decimal point must be included in the input stream, for either form, or an error message is given.

```
 0.0
 5.67E-4
-1.234001
 6E10
```

Screen 39

ENUMERATION INPUT/OUTPUT 5-8

The standard type BOOLEAN is an enumeration type and thus values of type BOOLEAN can be read in and written if a new package is declared.

```
package BOOLEAN_INOUT is new ENUMERATION_IO(BOOLEAN);
use BOOLEAN_INOUT;

ACTIVE                 :BOOLEAN;
...
GET(ACTIVE);
PUT(ACTIVE);
```

In the first example, the operator must type one of the words TRUE or FALSE at the keyboard. While in the second example, either TRUE or FALSE is displayed at the terminal.

User defined enumeration types are treated in a similar manner.

```
type COLORS is (RED,ORANGE,YELLOW,GREEN,BLUE,PURPLE);
package COLOR_INOUT is new ENUMERATION_IO(COLORS);
use COLOR_INOUT;
```

Screen 40

PARTS OF A PROGRAM 6-1

In simple terms, an Ada program consists of a specification part, a declarative part and a statement part. The specification of a program contains its name and a description of possible parameters to the subprogram. We shall come back to this later. In the declarative part, variables, constants and other parameters can be declared. The part of the program between the 'begin' and 'end' should contain a sequence of one or more statements. Each statement is executed once.

```
subprogram_specification is

   declarative part

begin
   statement 1;
   statement 2;
   .
   .
   .
   statement N;
end subprogram_name;
```

Screen 41

SIMPLE STATEMENTS 6-2

There are several kinds of statements, some are simple statements and some are compound statements. The most common simple statements are assignment statements and procedure calls; we have already seen examples of these. There is also a very simple statement called a null statement. When this statement is executed, nothing happens at all.

Here is an example showing a series of simple statements. The sequence reads in two real numbers and calculates their mean.

```
PUT_LINE("Enter two real numbers");
GET(X1);
GET(X2);
MEAN_VALUE := (X1 + X2) / 2.0;
PUT("The mean is: ");
PUT(MEAN_VALUE);
```

The statement:

```
MEAN_VALUE := (X1 + X2) / 2.0;
```

is an assignment statement and the others are procedure calls.

Screen 42

The most common compound statement is the if statement. The most common way of achieving selection in a program, that is, a choice between two or more different paths in a program, is to use an if statement. An if statement starts with the reserved word 'if' and terminates with the reserved words 'end if.' An if statement is comprised of a 'then' part followed by a number (possibly zero) of 'elsif' parts, ending possibly with an 'else' part.

When the statement is executed, the boolean expression that follow the words 'if' and 'elsif' are evaluated in order from the top down. If any of these boolean expressions are true, the sequence of statements in the corresponding part of the if statement is executed, and then control passes to the first statement after the words 'end if.' If all of the boolean expressions are false, but there is an 'else' part, then that sequence of statements will be executed. If there is no 'else' part, then the if statement terminates without any sequence of statements being executed.

```
if K > 5 or J < 4 then
   K := K + J;
   J := J + 1;
else
   K := J - K;
   K := K + 1;
end if;
```

```
if TEMPERATURE < 50.0 then
   PUT_LINE("Emergency!");
   RAD_SET := RAD_SET + 15.0;
elsif TEMPERATURE < 65.0 then
   PUT_LINE("Too Cold");
   RAD_SET := RAD_SET + 5.0;
elsif TEMPERATURE < 70.0 then
   PUT_LINE("OK");
```

Screen 43

CASE STATEMENT 6-4

We have seen how the if statement can be used to make a selection. A case statement can be used if a choice has to be made between several different alternatives. A case statement starts with the reserved word 'case' and ends with the reserved words 'end case.' After the word 'case', there is a discrete expression whose value determines the choice of one of the several alternatives. A discrete expression is an expression whose value is of a discrete type, that is, the expression is either an integer type or some enumeration type. A list of alternatives following the word 'when' is a list of the possible values that the discrete expression can assume.

```
type ONE_TO_FOUR is range 1..4;
NUMBER            :ONE_TO_FOUR;
...
case NUMBER
   when 1 =>
      PUT("ONE");
   when 2 =>
      PUT("TWO");
   when 3 =>
      PUT("THREE");
   when 4 =>
      PUT("FOUR");
end case;
```

Screen 44

If any possible values are omitted from the list of alternatives, there must be a special 'others' alternative. The 'others' alternative must come last in the case statement, so that when the case statement is executed, the 'others' alternative is reached only if the selector has a value other than those already enumerated in the earlier alternatives. The different alternatives in a list of alternatives can be enumerated with a vertical line or the interval containing them may be stated.

```
type MONTH_NUMBER is range 1..12;
MONTH              :MONTH_NUMBER;
...
case MONTH is
   when 1..2|12 =>
      PUT("WINTER");
   when 3|4|5 =>
      PUT("SPRING");
   when 6..8 =>
      PUT("SUMMER");
   when 9..11 =>
      PUT("AUTUMN");
   when others =>
      PUT("Error in month number");
end case;
```

Screen 45

LOOP STATEMENT 6-6

To perform iteration in Ada, that is, to execute one or several statements a number of times, a loop statement is used. There are three variations:

(1) a simple loop statement for writing part of a program that is to be executed an infinite number of times

(2) a loop statement with 'while', for writing part of a program that is to be executed a fixed number of times

(3) a loop statement with 'for', for writing part of a program that is to be executed until a certain condition is met

```
loop
   PUT_LINE("HELP! I can't stop");
end loop;

for INDEX in 1..10 loop
   PUT(INDEX);
end loop;

for NUMBER in reverse 1..5 loop
   PUT(NUMBER);
end loop;
```

```
while X > 1.0 loop
   PUT(X,FORE=>6,AFT=>2,EXP=>0);
   X := X / 2.0;
end loop;

while abs(NEXT_TERM) >= EPSILON loop
   SUM := SUM + NEXT_TERM;
   K = K + 1;
   SIGN := -SIGN;
   NEXT_TERM := SIGN / FLOAT(K * K);
end loop;
```

Screen 46

There is a special exit statement that can be used in conjunction with the loop statement. There are two variants, the first of which is simply:

```
exit;
```

This statement must lie within a loop statement. When it is executed, the iteration is terminated and control passes out of the loop statement to the first statement after 'end loop.'

The second variant of the exit statement is conditional. If the boolean expression is true, then a jump out of the loop statement takes place, just as in the simple exit described above. If the boolean statement is not true, execution continues with the next statement within the loop; no jump takes place.

```
loop
   if X > 10 then
      exit;
   else
      X := X + 2;
   end if;
end loop;
```

```
loop
   PUT("Enter data");
   GET(X);
   exit when X < 0.0;
   SUM := SUM + X;
   PUT(SUM,WIDTH=>3);
end loop;
```

Screen 47

The use of subprograms is a very important technique when mastering the complexity of program design. A program should normally be assembled from several subprograms, each of which describes a particular calculation or stage of the program. Subprograms can be thought of as building blocks that are used to construct a whole program.

In Ada there are two kinds of subprograms: functions and procedures. A function is used to describe the computation of a particular value and a procedure is used to describe an action that the program has to perform but that does not result in a direct value.

```
      procedure C (called by procedure B)

         function A (returns a value to procedure B)

      procedure B (calls function A and procedure C)

      procedure A (calls procedure B)

   main procedure (calls procedure A)
```

Screen 48

A function can be regarded as a 'black box' into which one or more values can be placed. Out of the box comes a result, whose value depends on the input values. As a simple example, we shall study a function MEAN_VALUE that calculates the mean of two floating point numbers.

```
function MEAN_VALUE(X1,X2:FLOAT) return FLOAT is
begin
   return (X1 + X2) / 2.0;
end MEAN_VALUE;
```

After the function's name, the data to be used in the function is specified by writing a list of the functions formal parameters. Two values of type FLOAT will be used in the function MEAN_VALUE.

After the reserved word 'return', the type of the result that will be returned by the function is specified. We have stated that the value returned by the function MEAN_VALUE will have type FLOAT.

```
            ____________
X ----->|              |
        | MEAN_VALUE   | -----> MEAN_VALUE of X and Y
Y ----->|______________|
```

Screen 49

```
function MEAN_VALUE(X1,X2:FLOAT) return FLOAT is
begin
   return (X1 + X2) / 2.0;
end MEAN_VALUE;
```

The function body describes what is inside the 'black box'. Included in the function body is an expression followed by the word 'return.' This expression should have the same type as specified after the word 'return' in the function specification. The value of the function will be the value of the expression. That is, when the return statement is executed, the computations in the 'box' terminate and what comes out of the 'box' is the value of the expression in the return statement. There can be several return statements in a function, but it is most common to have only one and for that one to be the last statement in the function.

A function is only a description of how a particular computation works, telling us what can go into the 'box' and what we can get out as a result. To invoke the computation, we have to put something into the 'box'; we must call the function. In the example that follows, we have put our function MEAN_VALUE into a complete program that reads in two numbers from the terminal, calculates their mean and displays it at the terminal.

Screen 50

```
with TEXT_IO;
package FLOAT_INOUT is new FLOAT_IO(FLOAT);
use TEXT_IO,FLOAT_INOUT;
procedure EVALUATE_MEAN is
   NUMBER1,NUMBER2,MEAN     :FLOAT;
   function MEAN_VALUE(X1,X2:FLOAT) return FLOAT is
   begin
      return (X1 + X2) / 2.0;
   end MEAN_VALUE;
begin
   PUT_LINE("Enter two real numbers");
   GET(NUMBER1);  GET(NUMBER2);
   MEAN := MEAN_VALUE(NUMBER1,NUMBER2);
   PUT("The mean is: ");  PUT(MEAN);
end EVALUATE_MEAN;
```

After the program has read the input values into NUMBER1 and NUMBER2, there follows a call to the function MEAN_VALUE. First, there is the name of the function being called and then, in brackets, there is a list of the actual parameters of the function. The values of the actual parameters are passed to the function. The formal parameters X1 and X2 are only temporary and only exist while the function call is in operation. The function call to MEAN_VALUE will take on the value calculated in the return statement of the body.

Screen 51

The other subprogram is the procedure. A procedure differs from a function in that it does not return a result when it is called. When a procedure is called, its sequence of statements is put into action. A procedure has the same form as a function. The only differences are that the reserved word 'procedure' is used instead of 'function', and that no result type is given in the procedure specification. In addition, since a procedure does not return any value as a result, there is not a return statement in the procedure. A procedure normally terminates when execution reaches the final end.

As an example, we shall write a procedure PRINT_CENTERED that will print any piece of text in the center of the line. This procedure will have a parameter of type STRING that gives the text to be printed.

```
procedure PRINT_CENTERED(TEXT:STRING) is
   LINE_LENGTH          :constant := 80;
begin
   SET_COL((LINE_LENGTH - TEXT'LENGTH) / 2);
   PUT(TEXT);
end PRINT_CENTERED;
```

Recall that 'LENGTH is an attribute of the type STRING which gives the number of characters in the string.

Screen 52

Note that the procedure does not return any value to the calling program. This is what distinguishes a procedure from a function. A procedure call is considered to be an entire statement in the calling program whereas a function call is considered to be an expression. A procedure call is terminated with a semicolon.

```
with TEXT_IO;
use TEXT_IO;
procedure PRINT_GREETING is

   procedure PRINT_CENTERED(TEXT:STRING) is
      LINE_LENGTH           :constant := 80;
   begin
     SET_COL((LINE_LENGTH - TEXT'LENGTH) / 2);
     PUT(TEXT);
   end PRINT_CENTERED;

begin
   NEW_LINE;
   PRINT_CENTERED("Hello");
   PRINT_CENTERED("I'm having a great time");
   PRINT_CENTERED("learning Ada");  NEW_LINE;
end PRINT_GREETING;
```

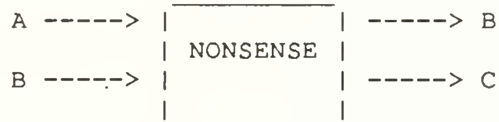
Screen 53

PARAMETER ASSOCIATION 7-7

In the previous example, we transferred text from the calling program to the subprogram PRINT_CENTERED. Transferring parameters between the calling program and the procedure can be carried out in many ways. We will use the following procedure to illustrate the principle of parameter association.

```
procedure NONSENSE(A:in INTEGER;B:in out INTEGER;C:out INTEGER) is
begin
   B := A + B;
   C := 0;
end NONSENSE;
```

In Ada, a parameter can be of mode in, mode in out, or mode out. In the procedure NONSENSE, these are exemplified by A, B and C, respectively. We can say that A is used to put values into the NONSENSE 'box', B is used both to put values in and get them out, and C is used only to get values out of the NONSENSE 'box'.



Screen 54

PARAMETER MODES 7-8

Recall that an actual parameter is the identifier passing a value from the calling procedure, while a formal parameter is the identifier receiving a value in the called procedure.

in ------ The actual parameter can be a variable, constant or an expression. It must have a value. The formal parameter is considered to be a constant that is initialized at the time of the call. The procedure is not permitted to change the value of a formal parameter. An in parameter may be given a default value when the formal parameter is specified.

out ----- The actual parameter must be a variable. The formal parameter can only be given a value. It can not be used in an expression.

in out -- The actual parameter must be a variable. It must have a value. The formal parameter can be used as an ordinary variable. Its value can be both used and changed. If the value is changed, the value of the actual parameter is changed.

If no mode is explicitly given in the parameter specification, the parameter is automatically an in parameter, that is, the specification is assumed.

Screen 55

PARAMETER DEMO 7-9

We shall put NONSENSE into a program that calls the procedure:

```
with TEXT_IO, BASIC_NUM_IO;
use TEXT_IO, BASIC_NUM_IO;
procedure PARAMETER_DEMO is
   X,Y,Z          :INTEGER;
   procedure NONSENSE(A:in INTEGER := 5;B:in out INTEGER;C:out INTEGER) is
   begin
      B := B + A;
      C := 0;
   end NONSENSE;
begin
   X := 1;  Y := 5; Z := 10;
   PUT(X);  PUT(Y);  PUT(Z);  NEW_LINE;
   NONSENSE(X,Y,Z);
   PUT(X);  PUT(Y);  PUT(Z);  NEW_LINE;
end PARAMETER_DEMO;
```

```
                  _____________________________________
            in   |                                     |  out
  X [1] -----> |  A := X   [1]                         | -----> Y [6]
        in out |  B := Y   [5] ---> B := B + A   [6]   | in out
  Y [5] -----> |  C := Z  [10] ---> C := 0       [0]   | -----> Z [0]
               |_____________________________________|
```

Screen 56

NAMED PARAMETER ASSOCIATION 7-10

The normal procedure for calling a subprogram is to list all the actual parameters, separated by commas. The first actual parameter is associated with the first formal parameter, the second actual parameter is associated with the second formal parameter and so on. This is called positional parameter association. In Ada there is another method of associating the actual parameters with the formal parameters in a subprogram call. It is possible to state the name of the formal parameters that the actual parameter is to be associated with. We call this named parameter association. When named parameter association is used, the parameters do not need to be listed in any special order. In addition, positional and named association can be mixed. When the two parameter associations are mixed in a call, the positional associations must be written first, in their correct order.

```
procedure MULTIPLE_WRITE(CHAR:CHARACTER;N:INTEGER := 2) is
begin
   for I in 1..N loop
      PUT(CHAR);  NEW_LINE;
   end loop;
end MULTIPLE_WRITE;
```

```
MULTIPLE_WRITE('X',3);                  MULTIPLE_WRITE(N=>3,CHAR=>'X');
MULTIPLE_WRITE(CHAR=>'X',N=>3);         MULTIPLE_WRITE('X',N=>3);
```

Screen 57

```
                    SCOPE OF A DECLARATION                          7-11

   It has been shown that a variable declared in a subprogram only exits while
the subprogram is called.  This means that the variable has only a certain
scope, which extends over the subprogram in which it is declared.  All sorts
of declarations, such as those of types, constants and subprograms, have an
associated scope so that what has been declared is only known and used in a
certain part of the program; it is said that they are only visible in that
part of the program.

                                 ___________________________________________
procedure P1 is                 | A declaration applies from the place where  |
   type T is ...;               | it is made to the end of the subprogram in  |
   A :constant:INTEGER := 100;  | which it is made.  This means that in our   |
   B :INTEGER := 2 * A;         | example, that the variable Q and the formal |
   procedure P2(X:INTEGER) is   | parameter I are only known in procedure P4. |
      C            :FLOAT;      | Any attempts to use Q or I outside of P4    |
      procedure P3(D:T) is      | will result in an error.                    |
         X         :FLOAT;      |---------------------------------------------|
      ...                       | A declaration that applies in a subprogram  |
   ...                          | P also applies in all the subprograms to P. |
   procedure P4(I:INTEGER) is   | The type T in our example is thus visible   |
      Q            :T;          | not only in P1, but also in P2 (and thus    |
   ...                          | also in P3) and in P4, because P2 and P4    |
...                             | are declared within P1.                     |
                                 ___________________________________________
```

Screen 58

In Ada, we have the option of compiling a program before internal subprograms are ready. We can tell the Ada compiler that we intend to write and compile a subprogram separately later. We do this by writing the word 'separate' after the subprogram specification.

```
with TEXT_IO;
use TEXT_IO;
procedure PRINT_GREETING is
   NAME     :STRING(1..10);
   procedure PRINT_NAME(NAME:
      in STRING) is separate;
begin
   PUT("Hi "); PRINT_NAME(NAME);
end PRINT_GREETING;
```

```
with TEXT_IO;
use TEXT_IO;
separate(PRINT_GREETING)
procedure PRINT_NAME(NAME:in STRING) is
begin
   PUT(NAME);
end PRINT_NAME;
```

The program is complete and can be compiled. Of course, testing is not possible until the separate subprograms are also written and compiled. When the subprograms are compiled separately, the program they are intended for must be specified. In front of the procedure name is written the word 'separate' followed by, in brackets, the name of the program in which it will be used. There is an advantage to using separate compilation in developing large programs. No program texts need to be very long and the program becomes easier to understand.

Screen 59

To declare several quantities with the same name in a subprogram is normally forbidden; but in Ada, to declare several subprograms with the same name in the same program is allowed and they are known as overloaded subprograms. Several subprograms may have the same name in a subprogram (or in a package) if they have different base types for parameters, and for the result in the case of functions.

```
type VECTOR is array(INTEGER range <>) of FLOAT;

function MEAN(X1,X2:FLOAT) return FLOAT is
begin
   return (X1 + X2) / 2.0;
end MEAN;

function MEAN(V:VECTOR) return FLOAT is
   SUM                    :FLOAT := 0.0;
begin
   for I in V'RANGE loop
      SUM := SUM + V(I);
   end loop;
   return SUM / FLOAT(V'LENGTH);
end MEAN;
```

Screen 60

OVERLOADED OPERATORS 7-14

In Ada, as we have seen, there are many built-in operators. The operator + exists for both integer and floating point types, for example, and the operator = is defined for all types we have encountered so far. When we declare our own types, we may also like to define operators for them.

```
   type VECTOR is array(INTEGER range <>) of FLOAT;
```

Assume we have declared variables of type VECTOR: X,Y,Z :VECTOR(1..5);
In order to add two variables, using a statement like Z := X + Y, we must define a new function "+" for identifiers of type VECTOR.

```
   function "+" (V1,V2:VECTOR) return VECTOR is
      TEMP             :VECTOR(V1'RANGE) := V2;
   begin
      for T in V1'RANGE loop
         TEMP(I) := TEMP(I) + V1(I);
      end loop;
      return TEMP;
   end "+";
```

We can say that we have overloaded operators in exactly the same sense as the overloaded subprograms we discussed earlier.

Screen 61

It has been shown that one subprogram can call another. Furthermore, a subprogram can call itself, and such a subprogram is called a recursive subprogram. It is appropriate to use recursive subprograms to solve certain types of problems. The problem for which recursion is most useful are those which are defined from the start in a recursive way; this occurs often in mathematical calculations. The most common example of a recursive subprogram is a function to calculate the factorial of a number n. The factorial of a number n, written n!, can be defined by:

$$n! = 1 \quad \text{if } n = 0;$$
$$n! = 1 \times 2 \times 3 \times \ldots \times n \quad \text{if } n > 0.$$

Another way of writing the definition is:

$$n! = 1 \quad \text{if } n = 0;$$
$$n! = n(n - 1)! \quad \text{if } n > 0.$$

The second definition leads naturally to the following Ada function:

```
function FACTORIAL(N:NATURAL) return POSITIVE is
begin
   if N = 0 then
      return 1;
   else
      return N * FACTORIAL(N - 1);
   end if;
end FACTORIAL;
```

Screen 62

RECORD TYPES 8-1

We have seen that by using array types we can describe complicated data objects with many components. One limitation of array types is that all the components of an array must be of the same kind. Therefore array types cannot be used to describe compound data objects where the components of an object are of different types. Instead, we use record types.

As an example, we shall study the description of a car in a hypothetical register of cars. If we have the type declarations:

```
type YEAR_TYPE is range 1900..2000;
type WEIGHT_TYPE is range 100..10000;  -- measured in kg
type POWER_TYPE is digits 4;           -- measured in kW
```

A definition of a record starts with the reserved word 'record' and ends with 'end record.' Between these words are declarations of the record type's components.

```
type CAR_TYPE is
   record
      REG_NUMBER            :STRING(1..7);
      MAKE                  :STRING(1..20);
      MODEL_YEAR            :YEAR_TYPE;
      WEIGHT                :WEIGHT_TYPE;
      POWER                 :POWER_TYPE;
   end record;
```

Screen 63

```
 _________________________________________________________
|              |       |              |         |         |
| REG_NUMBER   | MAKE  | MODEL_YEAR   | WEIGHT  | POWER   |     MY_CAR
|______________|_______|______________|_________|_________|
```

We can now define variables of type CAR_TYPE. MY_CAR :CAR_TYPE;
To access a particular component of a record, selection is used. A period is written after the name of the record, followed by the name of the component. For example, if we want to give the component WEIGHT in the variable MY_CAR the value 920, we can write the assignment MY_CAR.WEIGHT := 920;

Similarly:

```
MY_CAR.POWER := MY_CAR.POWER + 10.0;
if MY_CAR.POWER > 100.0 then
   PUT_LINE("Tuned");
end if;
PUT(MY_CAR.MAKE);
```

Selection is used for accessing the individual components of a record, but the whole record can also be handled at once.

```
YOUR_CAR := MY_CAR;
```

Screen 64

If we want to initialize the variable MY_CAR at the same time as declaring it, we write:

```
MY_CAR       :CAR_TYPE := ("C123XYZ","Ford Fiesta GL      ",1986,840,30.0);
```

We can also use record aggregates in assignments and comparisons, such as:

```
YOUR_CAR := ("ABD_544","Volvo 245 DL        ",1982,1400,70.0);

if YOUR_CAR = ("ABD_544","Volvo 245 DL       ",1982,1400,70.0) then ...
```

Only the comparison operators = and /= are defined. Thus it is not possible to see whether one record is 'greater than' or 'less than' another.

Named association can also be used by stating the name of the component and its corresponding value. In that case, it is not necessary to give the values of the components in the same order as they appear in the record.

```
YOUR_CAR := (REG_NUMBER => "ABD_544",
             MODEL_YEAR => 1982,
             MAKE => "Volvo 245 DL        ",
             POWER => 70.0,
             WEIGHT => 1400);
```

Screen 65

It is very common in real life to have a number of objects with the same properties. One example is the result list from a sporting event. For each competitor, his or her number, name, club and result are given. The natural data structure to use in Ada to describe such a real thing is an array of records. The result list from the sporting event could be described by:

```
type NUMBER is range 1..50;
type TIME is digits 7 range 0.0..600.0;
type COMPETITOR is
   record
      ID_NUMBER            :NUMBER;
      NAME                 :STRING(1..10);
      CLUB                 :STRING(1..20);
      RUN_TIME             :TIME;
   end record;

type RACE_LIST is array(1..50) of COMPETITOR;
```

Let's assume that the competitors are given ID_NUMBERs in the order in which they are ranked prior to the race. Specification is used to identify the particular components. For example, the final time of the person seeded 10th would be:

```
RACE_LIST(10).COMPETITOR.RUN_TIME
```

Screen 66

DYNAMIC OBJECTS 8-5

Once an array is declared, it is static. The number of components in the array cannot change during the execution of the program unit. However, in some applications, the number of objects needed is not known in advance. Then dynamic data structures are required, that can grow and shrink during program execution; it must therefore be possible to create new objects during execution. An example of a dynamic data structure is a list where elements can be added and removed dynamically.

To create a dynamic object during execution, an allocator is used. This, in its simplest form, is an expression with the reserved word 'new' followed by a type name. To create a dynamic object of type INTEGER, for example, we can write

```
new INTEGER;
```

It is most common to work with dynamic objects of record types. If, for example, we have declared the type PERSON, we can then create a new object of type PERSON using the allocator.

```
type PERSON is
   record
      NAME      :STRING(1..20);
      SSN       :STRING(1..9);
   end record;
new PERSON;
```

Screen 67

ACCESS VARIABLES 8-6

It is possible to add an initialization term to the allocator to give a newly created object a value. After the name, an apostrophe is written followed by the values in brackets. The values in brackets do not need to be constants, but they must at least be expressions with the same type as the new object.

```
new INTEGER'(5);
new INTEGER'(2 * N);
new PERSON'("John Smith          ","123456789");
```

The result of an allocator is a reference or a pointer, or, as generally called in Ada, an access value. We say that this result is of type ACCESS; in Ada, the reserved word 'access' is used to denote this pointer type.

As for other types, it is possible to declare variables of ACCESS types. In the following example, two access variables are declared, P1 and PP.

```
type INT_POINTER is access INTEGER;
type PERS_POINTER is access PERSON;

PI            :INT_POINTER;
PP            :PERS_POINTER;
```

Screen 68

POINTERS 8-7

We can now use access variables to save the pointers to the created objects.

```
PI := new INTEGER'(5);            PI  ----> |   5    |
                                            |________|

PP := new PERSON'(John Smith          ","123456789");


     PP ----> | John Smith                |123456789|
              |___________________________|_________|
                        NAME                  SSN
```

If an access variable is not initialized with its declaration, it automatically takes the value null which means it is not pointing to anything.

```
PI := new INTEGER;       PI ---->

PP := new PERSON;        PP ---->
```

Screen 69

ACCESSING AN OBJECT 8-8

Now the pointer can be used to get at the new object -- it points to it; it provides access to it. To change a person's name, for example, we can write:

```
PP.NAME := "Johnathan P Smith ";
```

and to write out the person's SSN, we can make the call

```
PUT(PP.SSN);
```

If the whole object that PP points to is required, the reserved word 'all' can be used.

```
PP.all := ("Jane Smith          ","987654321");
```

Several access variables may point to the same object. If we declare another variable

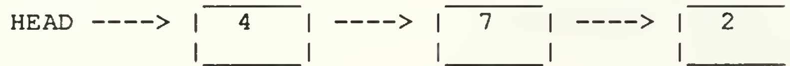
```
PP2          :PERS_POINTER;
```

We can now write the statement PP2 := PP;

```
              ______________________________
 PP  ----> | Jane Smith          |987654321|
PP2  ----> |_____________________|_________|
                  NAME               SSN
```

Screen 70

A linked list, or simply a list, is a dynamic data structure with applications in many different areas of programming. A list of three integers is illustrated. Each element in the list contains a value and a pointer to the next element in the list. The first element in a list is usually called its head and that element is pointed to by a special pointer.



If we put the declaration of LINK before that of LIST_ELEMENT, then LIST_ELEMENT will be undefined when LINK is declared. Therefore, we use an incomplete type declaration where it is only stated that LIST_ELEMENT is a type. Later, a full type declaration is made.

```
type LIST_ELEMENT;
type LINK is access LIST_ELEMENT;
type LIST_ELEMENT is
   record
      VALUE       :INTEGER;
      NEXT        :LINK;
   end record;
```

Screen 71

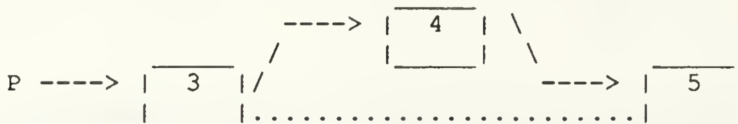
```
                     BUILDING UP A LIST                            8-10

   We start building up a linked list by declaring an access variable:

             LINK          :LINK;                 LIST ---->

   This variable automatically takes the value null on declaration, which
describes the fact that the list is empty.  We can create a new element and
add it to the list.                                                 ______
             LIST := new LIST_ELEMENT'(2,LIST);   LIST ----> |  2   |
                                                             |______|
   The second part of the list element, the pointer to the next element in the
list, automatically gets the value null when the element of the list is
created.  Assume that we now want to create other elements in the list
containing the values 7 and 4.

   procedure PUT_FIRST(DATA:in INTEGER;L:in out LINK) is
   begin
      L := new LIST_ELEMENT'(DATA,L);       -- The new element is created first
   end PUT_FIRST;                           -- then the pointer L is moved

We can build up the previous list by making the calls
                                         ______        ______        ______
   PUT_FIRST(7,LIST);        LIST ----> |  4   | ----> |  7   | ----> |  2   |
   PUT_FIRST(4,LIST);                   |______|       |______|       |______|
```

Screen 72

ADDING AND DELETING ELEMENTS 8-11

It is sometimes necessary to place a new element in a particular position in a linked list. Suppose, for example, we have an element in a list and a pointer P is pointing to it. Now suppose we want to put a new element with value 4 after this. This new element can then be created and placed in the list with a single statement.

```
P.NEXT := new LIST_ELEMENT'(4,P.NEXT);
```



If we wanted to insert a new element in front of a particular element that is pointed to, it would be much more trouble. Then we would need to run through the list from the beginning in order to get access to the element in front of the one pointed to.

In certain cases, it is simple to remove elements from a list. The first element can be removed with the statement `LIST := LIST.NEXT;`

It is also easy to remove the element coming after one to which there is a pointer. For example, we can remove the element that lies after the one that P points to.

```
P.NEXT := P.NEXT.NEXT;
```

Screen 73

RUNNING THROUGH A LIST 8-12

We shall study a function that searches for a particular element in a list. If we are searching for a particular person in a list of people, the function returns the pointer to the corresponding element in the list; otherwise, it will return a value of null.

```
type PERSON;
type PERSON_LINK is access PERSON;
subtype NAME_TYPE is STRING(1..20);
type PERSON is
   record
      NEXT          :PERSON_LINK;
      NAME          :NAME_TYPE;
      LENGTH        :INTEGER;
      WEIGHT        :FLOAT;
   end record;
```

```
function FIND_PERSON(L:PERSON_LINK;
   REQ_NAME:NAME_TYPE)
   return PERSON_LINK is
   P            :PERSON_LINK := L;
begin
   while P /= null and then P.NAME
      /= REQ_NAME loop
      P := P.NEXT;
   end loop;
   return P;
end FIND_PERSON;
```

In the function, the pointer P is made to run all through the list until it is finished (P has the value null) or until the required person is found.

Screen 74

EXCEPTIONS 9-1

When a program is executed, unexpected situations sometimes occur. Such a situation is called an exception. The exception may be the result of an error of some kind, for example, dividing by zero, using an index to an array outside the allowed constraints, or giving faulty input data to a program or subprogram. An exception is not necessarily the result of an error in the program. It may be something that only happens very rarely when the program is run. In Ada there are five predefined exceptions:

CONSTRAINT_ERROR	Occurs, for example, if a variable is assigned an illegal value or indexing is attempted beyond the limits of an array.
NUMERIC_ERROR	Occurs when a numeric operation cannot give a correct result. A common case is trying to divide by 0.
PROGRAM_ERROR	Occurs in unusual circumstances, when part of the program called is not accessible, or when the final end in a function is reached, that is, the function has not returned a result.
STORAGE_ERROR	Occurs if the accessible memory expires, for example, there is a recursive subprogram with a faulty terminating condition so that too many instances of the subprogram are generated.
TASKING_ERROR	Occurs in connection with parallel programs.

Screen 75

DECLARING EXCEPTIONS 9-2

When an exception occurs, the normal execution of the program ceases immediately. If no special precautions have been taken in the program, it will terminate abnormally with an error message stating the type of exception causing the termination.

In an Ada program, it is possible to work with exceptions other than the five predefined ones listed previously. It is possible to define our own exceptions. An exception TIME_UP, for example, could be declared as follows:

```
TIME_UP   :exception;
```

This is a declaration and is placed with the other declarations -- declarations of variables, for example. In form, it looks like a variable declaration, but TIME_UP is not a variable. It cannot have a value. All the declaration says is that in the program there is an exception, TIME_UP, that may happen. We can make as many declarations as we like of our own exceptions in a program. We can list several in one declaration, for example:

```
TABLE_EMPTY, TABLE_FULL       :exception;
```

To cause an exception to occur, we can use a raise statement: raise TIME_UP; Unless we do something special, normal execution of the program ceases with an error message:

** MAIN PROGRAM ABANDONED -- EXCEPTION "time_up" RAISED **

Screen 76

HANDLING EXCEPTIONS 9-3

So far we have said that an exception interrupts a program so that it stops with an error message. It is, however, possible to trap exceptions in a program and take some appropriate action. If we have a program that controls a critical process, it is not acceptable for the program to cease abruptly if an exception occurs. The program must deal with what has happened.

There are three levels of ambition in dealing with exceptions:

(1) Take control of the exception, and try to take suitable action to enable the program to continue.

(2) Trap, identify and pass the exception on to another part of the program.

(3) Ignore the exception, that is, the program will stop when the exception occurs.

The basic principle should be that the exception is controlled in the part of the program (or outside it, if necessary) where its effect can most sensibly be handled.

Screen 77

CONTROLLING AN EXCEPTION 9-4

As an example of controlling an exception, we shall study a function that calculates the tangent of an angle. We assume that we have access to a mathematical package containing the functions SIN and COS, but not TAN.

```
function TAN(X:FLOAT) return FLOAT is
begin
   return SIN(X) / COS(X);

exception
   when NUMERIC_ERROR =>
      if (SIN(X) >= 0.0 and COS(X) >= 0.0) or
         (SIN(X) < 0.0 and COS(X) < 0.0) then
         return FLOAT'LARGE;
      else
         return -FLOAT'LARGE;
      end if;
end TAN;
```

We have inserted the part that starts with the reserved word 'exception' to deal with an attempt to divide by 0. When the execution of the return SIN(X)/COS(X) statement raises a NUMERIC_ERROR, control passes to the statements that are in the appropriate exception handler. Control subsequently returns to the program that called TAN and execution continues as normal.

Screen 78

There can be handlers for several exceptions at the end of a subprogram, both predefined and declared.

```
exception
   when TIME_UP =>
      PUT_LINE("Time to make a move");
   when CONSTRAINT_ERROR =>
      PUT_LINE("Error in index value");
   when TABLE_FULL | TABLE_EMPTY =>
      PUT_LINE("Table error");
   when others =>
      PUT_LINE("Something wrong in package PLAY");
```

We see that it is possible to have common handlers for two or more kinds of exceptions. In our example, TABLE_FULL and TABLE_EMPTY have a common handler. Thus, the message "Table error" is output when exceptions of the kind TABLE_FULL or TABLE_EMPTY occur. We also see that there can be an others alternative to handle all other kinds of exception that are not already listed. An others alternative must appear at the end, if it occurs at all. If an exception occurs and the current subprogram has no handler for exceptions of this kind, then the exception is 'passed on' to the subprogram that called the current subprogram. If this also has no exception, it is passed on further. If no handler has been encountered at the top, an error occurs.

Screen 79

TRAPPING AN EXCEPTION 9-6

We can now write several lines to trap the exception NUMERIC_ERROR. This then means that the program is not interrupted and can continue by asking for another value.

```
loop
   begin
      PUT("Enter a real number, or CTRL-D to quit: ");
      exit when END_OF_FILE;
      GET(NUMBER);
      RES := TAN(NUMBER);
      PUT("Tangent is: ");  PUT(RES);  NEW_LINE;
   exception
      when NUMERIC_ERROR =>
         PUT_LINE("No tangent can be evaluated");
   end
end loop;
```

Note the words 'begin' and 'end' around the contents of the loop statement. This is a block statement. An exception may be placed at the end of a block statement. When an exception occurs within the block, control passes to the appropriate handler (if there is one). When the statements within the handler have been executed, execution of the whole block statement stops and the program continues with the next statement after the block statement.

Screen 80

FILES 10-1

All programs we have studied so far have read from or written to the terminal. However, it is often necessary for a program to work with other external devices connected to the computer. For example, output might be required on a line printer or a special, high-quality printer instead of the terminal. Another problem is that the variables used in a program only exist while the program is being executed. If the data is to be saved permanently, so that it survives when program execution has finished, it must be stored in the computer's secondary storage, most often on disk. It is therefore important to be able to read and write data to and from secondary storage.

In computer jargon, an arbitrarily long sequence of data objects where all the objects have the same type is called a file. A file can be stored in the computers secondary storage, for example, on disk, and can therefore be used to store data permanently. If these individual objects in a file have the type CHARACTER, the file is called a text file. In contrast, if the individual objects do not have the type CHARACTER, the file is usually called a binary file. These objects are actually represented in the same binary form used internally in a program.

In Ada, there are two basic categories of files: sequential files where objects must be read and written in their correct order from start to finish; and direct access files or direct files, where an arbitrary object can be accessed without going through the file in a particular order.

Screen 81

TEXT FILES 10-2

Text files belong to the category of sequential files. A text file is understood logically, by a program, as a series of characters to be either read or to be written. Each text file exists physically in the computer system and has a special name. The rules for the format of such names do not need to agree (and generally do not agree) with the rules for identifiers in Ada. The format of the name depends entirely on the computer system in use.

In an Ada program, all work is performed on logical text files. In the package TEXT_IO, there is a type FILE_TYPE that can be used to declare such logical text files. At the start of a program, we can declare a logical text file (a file variable), for example: INFILE :FILE_TYPE;
We shall distinguish between a logical file in a program and a physical file in the computer system, by calling the formal simply a file and the latter an external file.

Before reading or writing a file can begin in a program, it must be connected with an external file. This is done by calling either CREATE or OPEN in TEXT_IO. CREATE is used when a new external file is to be created and OPEN is used when work is to be performed on an existing external file. Both CREATE and OPEN take parameters for the logical file name, the file mode (either IN_FILE to read or OUT_FILE to write) and the external file name.

When reading from or writing to a file is finished, the file has to be closed by calling the procedure CLOSE. The only parameter necessary is the file name. If we forget to close a file in a program, the result is not well defined. Therefore, we should get into the habit of closing a file after use.

Screen 82

An open file can either be read from or written to, and all the subprograms previously discussed in connection with reading from and writing to the terminal, such as GET, PUT and NEW_LINE. The only difference is that a file has to be given as a first parameter in any call that is not destined for the terminal.

```
with TEXT_IO;
use TEXT_IO;
procedure COPY_FILE is
   INFILE, OUTFILE         :FILE_TYPE;
   LINE                    :STRING(1..200);
   LINE_LENGTH             :NATURAL;
begin
   OPEN(INFILE,MODE=>IN_FILE,NAME=>"old.file");
   CREATE(OUTFILE,NAME=>"copy.file");  -- default mode is OUT_FILE
   while not END_OF_FILE(INFILE) loop  -- END_OF_FILE returns a boolean
      GET_LINE(INFILE,LINE,LINE_LENGTH);
      PUT_LINE(OUTFILE,LINE(1..LINE_LENGTH));
   end loop;
   CLOSE(INFILE);
   CLOSE(OUTFILE);
end COPY_FILE;
```

Screen 83

When we want to work with sequential files other than text files, we shall make use of another standard package called SEQUENTIAL_IO. To gain access to this package, the following line must appear at the start of the program.

```
with SEQUENTIAL_IO;
```

In a text file the objects are always of type CHARACTER, but in the case of other sequential files, the objects can be of any type. For this reason, SEQUENTIAL_IO is not a ready-made package like TEXT_IO. It is a generic package (template) that can be used to tailor input/output packages for sequential files in which the objects have exactly the required type.

Other than text files, the most common files to work with are those in which the objects have a record type. Assume we want to collect information about the social security numbers of a group of people. We can store the data in a file in which each object is a record containing the person's name and social security number. Other information can also be maintained, however, input and output packages must be instantiated to deal with the numeric types.

```
type PERSON is
   record
      NAME          :STRING(1..20);
      SSN           :STRING(1..9);
   end record
```

Screen 84

SEQUENTIAL PACKAGE INSTANTIATION 10-5

To create a package containing the facilities for handling files in which the objects have type PERSON, we write:

```
package PERSON_INOUT is new SEQUENTIAL_IO(PERSON); --instantiate
PERSON_INOUT
```

The word PERSON in brackets states that the objects will be of type PERSON. The new package will be called PERSON_INOUT. As in the case of text files, logical files are worked with in a program and file variables are declared to represent the logical files. To declare a logical file in which the objects have the type PERSON we can use our new package PERSON_INOUT and write:

```
PERSON_FILE          :PERSON_INOUT.FILE_TYPE;
```

Note that we have used selector notation and written PERSON_INOUT.FILE_TYPE. We could have included a use PERSON_INOUT clause and only write FILE_TYPE, but the compiler would not have known if we meant the FILE_TYPE specified in TEXT_IO or the FILE_TYPE specified in PERSON_INOUT. Therefore, selector notation is generally used for all references other than to TEXT_IO and additional use clauses are not necessary.

Screen 85

READING/WRITING SEQUENTIAL FILES 10-6

To link a logical text file in a program with a physical file, the procedures CREATE and OPEN are used. These work exactly the same way for general files as for text files. Reading and writing are in some respects simpler for general sequential files. GET and PUT are not used, but the two procedures READ and WRITE are specified in SEQUENTIAL_IO. The parameters required for both READ and WRITE are the logical file name and the item to be read. For example, assume we have declared a record variable P

```
P            :PERSON;
```

and we have opened the file PERSON_FILE to read from it. We can then make the call:

```
READ(PERSON_FILE,P);
```

This means that a record in PERSON_FILE is read and copied to P. Note that the entire record is read from a file at once. It is not possible to read an individual element of data; it is not possible to read only the name from a record in the file, for example. This demands that the types of the objects in the file and the variable being read to are the same.

Reading and writing cannot proceed at the same time in sequential files. There is, however, the possibility of resetting -- going back to the beginning of the file and using it in a new way. The RESET procedure takes the logical file name as its only parameter.

Screen 86

It is not necessary to handle the records in a file sequentially in an Ada program. If instead of using the package SEQUENTIAL_IO another standard package, DIRECT_IO, is used, it becomes possible to read and write records in an arbitrary order. For every open file the Ada system keeps a current index which points to the next record waiting to be read. Work with direct files offers the possibility of controlling the index. In the package DIRECT_IO, there are the type declarations:

```
type COUNT is range 0..implementation_dependent_integer;
subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;
```

The current index has the type POSITIVE_COUNT. To set the current index to a particular value, the procedure SET_INDEX is used. The parameters for SET_INDEX are the logical file name and the record number that you want to set the index pointer to. The value of the current index can be read by calling the function INDEX. The only parameter required is the logical file name. To determine how many records there are in a file (that is, the index number of the last record) the function SIZE can be used. The only parameter required is the logical file name. INDEX returns values of type POSITIVE_COUNT, while SIZE returns values of type COUNT (there is not an index value of 0, but there can be 0 records).

Screen 87

When a record is to be read or written, it is possible to state which record is required. There are two versions of the subprograms READ and WRITE in the package DIRECT_IO. The first pair is identical to the READ and WRITE found in SEQUENTIAL_IO. Thus, direct files can be treated in exactly the same way as sequential files.

The second version of the two procedures has a third parameter FROM in READ and TO in WRITE. This third parameter specifies the index in the file for the record that is to be read or written. Before reading or writing starts, the current index is set to the value given. To read record number 100 of file F, for example, we can write:

```
READ(F,R,100);
```

For all versions of READ and WRITE, the current index is automatically increased by 1 after reading or writing. For example, after the above call, the current index has the value 101.

As for other files, a direct file must be created or opened before reading or writing can begin. However, a direct file can be opened for both reading and writing. The file should be specified of type INOUT_FILE when the file is opened. For the procedure CREATE, the MODE parameter may be omitted because the default value is INOUT_FILE. The remaining subprograms in SEQUENTIAL_IO are also found in DIRECT_IO and work in exactly the same way.

Screen 88

PACKAGES 11-1

Subprograms, types and objects that logically belong together in some way can be put together in a package. When a package is constructed, its interface with the rest of the program, in other words, the part of the package that will be visible to the program, has to be specified. Details that are not essential to the user of the package can then be concealed within the package. A package can be developed and compiled alone.

When a complicated product, such as a car, is being built, it is necessary to make the different parts separately in order to prevent the work from becoming too complex. Eventually, the separate parts are assembled into a complete product. To fit the parts together successfully, a specification of how the parts fit must have been carefully made during the design phase.

Ada's package facility allows a program to be built up in the form of several separate modules, each of which forms a logical unit. With the help of a package specification it can be state how the package should be put together with other parts of the program. Working with large, unmanageable programs is thus avoided: one subprogram can be tackled at a time.

It is also possible to build up a library of general packages that may be used in several contexts within different programs. These could include a package of different mathematical functions or a package of tools for presenting results in graphical form. These packages may have been written by the individual programmer, or have been obtained from another source.

Screen 89

PACKAGE SPECIFICATIONS 11-2

Each package has a specification. This can be regarded as the package's 'shop window' that says what the package has to offer the potential user. The specification specifies the package interface with other parts of the program. A package specification is introduced with the reserved word 'package' followed by the package name. Within the specification, declarations of types and objects can be made and several subprograms' specifications can be given.

Suppose we want to work with ordinary two-dimensional geometric figures, such as rectangles, circles and triangles. It may then be appropriate to construct a package containing the tools for performing various calculations on these figures. We can call it PLANIMETRY and write a specification for it. In addition to defining types LENGTH and AREA that describe lengths and areas associated with the figures, we shall declare functions that calculate the areas of rectangles, circles and triangles.The function bodies themselves will be contained in the body.

```
package PLANIMETRY is
   type LENGTH is digits 5 range 0.0 .. 1.0E10;
   type AREA is digits range 0.0 .. 1.0E20;
   function AREA_RECTANGLE(L,H:LENGTH) return AREA;
   function AREA_CIRCLE(R:LENGTH) return AREA;
   function CIRCUM_CIRCLE(R:LENGTH) return LENGTH;
   function AREA_TRIANGLE(B,H:LENGTH) return AREA;
end PLANIMETRY;
```

Screen 90

ORDER OF COMPILATION 11-3

Before we go on to see how a package can be used and what a package body looks like, a few words must be said about the programming environment in Ada. When a program or a part of a program is compiled, a compiler is used. The compiler reads the program text and gives the program translated into machine code as a result. Compilers of all kinds work in this way, not only the Ada compiler.

An Ada compiler differs from most other language compilers, however, in that it not only produces machine code but also keeps track of all the compilation that is performed. The compiler maintains what is called an Ada library. When a compilation is complete, the Ada compiler puts a description of the program (or part of a program) that has been compiled into the Ada library.

This means that it is possible to refer to what has been compiled earlier in a program. The compiler goes into the Ada library and searches for information about the relevant item, making it feasible to build large, complicated programs gradually. During compilation, a compilation unit is fed into the compiler. In a compilation unit, such as a procedure, a 'with' clause can be put first, enabling reference to be made to other compilation units. If we compile a procedure, for example, we can have the clause:

```
with Q,R:
```

This means that Q and R need to have been compiled earlier.

Screen 91

PACKAGE BODIES 11-4

Now we shall look at how to construct a package body -- the part of the package that is concealed from the user. Details that the user does not need to know are placed in the package body; for example, the subprogram bodies and internal data.

A package body is introduced with the reserved words 'package body.' The rest of the package body has the same structure as a subprogram body. First comes a declarative part and then a sequence of statements. This latter section can be omitted, and this is most common.

```
package body PLANIMETRY is

   PI    :constant := 3.14159;

   function AREA_RECTANGLE(L,H:LENGTH)
            return AREA is
   begin
      return AREA(L) * AREA(H);
   end AREA_RECTANGLE;
   function AREA_CIRCLE(R:LENGTH)
            return AREA is
   begin
      return P1 * AREA(R) ** 2;
   end AREA_CIRCLE;

   function CIRCUM_CIRCLE(R:LENGTH)
            return LENGTH is
   begin
      return 2.0 * PI * R;
   end CIRCUM_CIRCLE;

   function AREA_TRIANGLE(B,L:LENGTH)
            return AREA is
   begin
      return AREA(B) * AREA(H) / 2.0;
   end AREA_TRIANGLE;

end PLANIMETRY;
```

Screen 92

RELATIONSHIP BETWEEN SPECS AND BODY 11-5

Within the package body are the complete function bodies for the functions that were declared in the package specification. There is also a constant PI and this is only known within the package body. Thus, in a program that uses the package PLANIMETRY, it is not permitted to write:

```
PLANIMETRY.PI    -- ERROR! PI is only know within the package body
```

It is only the items from the package SPECIFICATION that are know outside.

A package body must be compiled after its specification has been compiled. In the specification we talk about what the package will be able to do and in the body we state how to do what we have promised. It is possible to compile the specification and body separately on separate occasions or, if the texts are to remain together, to compile then as two compilation units but fed to the compiler together. In the latter case the specification should come before the body. It is recommended that the two parts be compiled separately. The package body contains whole programs which are likely to be amended recompiled many times during program development. It is therefore advantageous to separate the specification and the body to prevent having to recompile the package specification at the same time. However, if the specification is recompiled, all the programs using the package must also be recompiled.

Screen 93

DIFFERENT CATEGORIES OF PACKAGES 11-6

Packages can be used for different purposes when a Ada program is written. Generally, there are four different categories of packages:

(1) Packages with a collection of types and constants, for example, a package of mathematical constants

(2) Packages with a group of subprograms that logically belong together, for example, a package of standard mathematical functions

(3) Packages with 'memory' that can be used to represent complicated objects in different states

(4) Packages with construct abstract data types

Our package PLANIMETRY belongs to category (2). The following package falls into category (1). Note that no body is required for packages of this type.

```
   package ATOMIC_CONSTANTS is
      ELECTRONIC_CHARGE      :constant := 1.602E-19;    -- coulomb
      ELECTRON_MASS          :constant := 0.9108E-30;   -- kg
      NEUTRON_MASS           :constant := 1674.7E-30;   -- kg
      PROTON_MASS            :constant := 1672.4E-30;   -- kg
   end ATOMIC_CONSTANTS;
```

Screen 94

DIFFERENT CATEGORIES OF PACKAGES (CON'T) 11-7

The special thing about packages of category (3), packages with memory, is that they can be in different states. Each time the package is used, its state changes. Therefore, the package must be able to 'remember' its state between uses.

The standard package TEXT_IO can be said to belong to category (3). It represents the real object -- the 'terminal'. The package must remember how long a line of output can be and how many lines there can be on a page of output. When printing output, between calls to PUT, PUT_LINE and NEW_LINE, the package must remember which page is being printed, which line it is on and how much of the output has already been printed.

Earlier, we talked about a data type being characterized by the values that its objects can assume and by the operations that can be carried out on them. For complicated types that we construct ourselves, such as array and record types of various kinds, only comparison operators are normally automatically defined, but we can create new operators for such types by writing subprograms and combining them into a package of category (4). For example, we could create a package to perform addition, subtraction, multiplication and division on complex numbers.

```
type COMPLEX_NUMBER is
   record
      REAL,IMAGINARY   :FLOAT;
   end record;
```

Screen 95

PRIVATE TYPES 11-8

We have seen that it is possible to hide away in the body of a package those details that are of no interest to a user. There is another possible way of concealing the details of an abstract data type in Ada, using so-called private types. Within the body of a package, the package's private types may be used freely, however, outside the body of the package the only operations allowed are assignment and comparison for equality and inequality. There are occasions when the package designer does not want the package user to be able to carry out even these operations on a private type. Then a limited private type can be declared. The reserved word 'limited' signifies precisely that assignment and comparison are forbidden.

It is the limitation in manipulation of the abstract data types that makes the use of private types so valuable. For example, a stack can be represented as either an array or as a linked list. The package specification for both is identical except for the private types. Because the data types are different, the package bodies will be significantly different.

standard operations for stack manipulation:

```
   procedure CREATE(S:in out STACK);
   procedure PUSH(A:ATOM,S:in out STACK);
   procedure POP(S:in out STACK;A:in out ATOM);
   function EMPTY(S:STACK) return BOOLEAN;
```

Screen 96

```
                    ARRAY IMPLEMENTATION OF STACK                    11-9

package STACK_ADT is
   type ATOM is . . . ;      -- can be a value or a record of values
   type STACK is private;

   -- procedure and function declarations go here

private
   MAXSIZE         :constant INTEGER := ...;  -- size depends on
implementation
   subtype POSITION is INTEGER range 0 .. MAXSIZE;
   type DATA_ARRAY is array(1 .. MAXSIZE) of ATOM;
   type STACK is record
      TOP      :POSITION;
      DATA     :DATA_ARRAY;
   end record;
end STACK_ADT;

initial stack:             PUSH g:                      POP:

TOP  6                     TOP 7                        TOP 6
DATA a b c d e f          DATA a b c d e f g           DATA a b c d e f
     1 2 3 4 5 6 ... MAX       1 2 3 4 5 6 7 ... MAX        1 2 3 4 5 6 ... MAX
```

Screen 97

```
                    POINTER-BASED IMPLEMENTATION OF STACK            11-10

package STACK_ADT is
   type ATOM is . . . ;      -- can be a value or a record of values
   type STACK is private;

   -- procedure and function declarations go here

private
   type NODE;
   type STACK is access NODE;
   type NODE is record
      A         :ATOM;
      NEXT      :STACK;
   end record;
end STACK_ADT;


 initial      |.|a| <-- |-|b| <-- |-|c| <-- |-|d| <-- |-|e| <-- |-|f|
  stack

  PUSH g      |.|a| <-- |-|b| <-- |-|c| <-- |-|d| <-- |-|e| <-- |-|f| <-- |-|g|


  POP         |.|a| <-- |-|b| <-- |-|c| <-- |-|d| <-- |-|e| <-- |-|f|
```

Screen 98

When a program or part of a program is being written, it is usually advantageous to try to make it as general as possible. Then, if the conditions for a program should change, fewer changes (or possibly none) will be required to enable the program to work. Moreover, similar programming problems occur in many different contexts. If a general solution has been designed for one problem, it can often be used on other problems later.

Ada offers the programmer the possibility of writing general programs using generic units. Such a program unit can be either a subprogram or a package. A generic unit is not only one subprogram or one package, but is a description of a whole family of similar units. Generic units can be regarded as generalized bits of a puzzle that can be fitted together to develop a new program.

The following procedure demands that the two parameters have the type INTEGER so it cannot be used for swapping the values of, say, two floating point variables.

```
procedure SWAP(NUMBER1,NUMBER2:in out INTEGER) is
   TEMP              :INTEGER;
begin
   TEMP := NUMBER1;
   NUMBER1 := NUMBER2;
   NUMBER2 := TEMP;
end SWAP;
```

Screen 99

We shall now rewrite SWAP so that it can be used for all types of parameter. It then gets the new specification.

```
generic                          -- specification must be compiled before body
   type ELEMENT is private;
procedure SWAP(A,B:in out ELEMENT);
```

Between the reserved words 'generic' and 'procedure' is a list of the format generic parameters. In this case there is only one such parameter, the type ELEMENT. The procedure must also have a body:

```
procedure SWAP(A,B:in out ELEMENT) is
   TEMP              :ELEMENT;
begin
   TEMP := A;  A := B;  B := TEMP;
end SWAP;
```

This looks like a perfectly normal procedure, but its not. It is a template that defines a family of procedures. In the body of SWAP, a generic parameter ELEMENT is used. SWAP describes different procedures depending on the value of ELEMENT. If ELEMENT has the value FLOAT, for example, SWAP describes a procedure that can be used to interchange the values of two variables of type FLOAT. Think of the word FLOAT replacing all instances of the word ELEMENT.

Screen 100

If the procedure SWAP is to be used in a program, the program should begin: with SWAP;

So far there is no 'real' procedure SWAP; there is only a template. To get a procedure, we have to use the template and generate or create a version of SWAP. A particular version of a generic unit is called an instance of the generic unit and is created by making a generic instantiation. When an instance of a generic unit is created, the value of the generic parameter has to be specified: the actual generic parameters are stated. To instantiate a procedure to interchange two numbers of the type FLOAT, for example, we make the declaration:

```
procedure SWAP_FLOAT is new SWAP(FLOAT);
```

Here FLOAT is the actual generic parameter. Now we have a true procedure that can be called in the normal way. If the variables X and Y have the type FLOAT:

```
SWAP_FLOAT(X,Y);
```

If we also want a procedure that can swap two character variables, we can instantiate another instance of the procedure SWAP:

```
procedure SWAP_CHAR is new SWAP(CHARACTER);
```

Screen 101

It is also possible to have a generic package. To illustrate this, we shall give a simple version of the stack package discussed earlier.

```
generic
   type ELEMENT is private;
package STACK is
   procedure PUSH(E:in ELEMENT);
   procedure POP(E:out ELEMENT);
   function EMPTY return BOOLEAN;
end STACK;

package body STACK is
   SIZE       :constant := 100;
   A          :array(1..SIZE) of ELEMENT;
   TOP        :NATURAL := 0;

procedure PUSH(E:in ELEMENT) is
begin
   TOP := TOP + 1;
    A(TOP) := E;
end PUSH;

procedure POP(E:out ELEMENT) is
begin
   E := A(TOP);
   TOP := TOP - 1;
end POP;

function EMPTY return BOOLEAN is
begin
   return TOP = 0;
 end EMPTY;
```

Just as with generic procedures, new instances can be created as needed:

```
   package INT_STACK is new STACK(INTEGER);
   package PERS_STACK is new STACK(PERSON);
```

Screen 102

TASKING 12-1

The final major topic to be introduced is tasking. This has been left to the end, not because it is important or particularly difficult, but because, apart from the interaction with exceptions, it is a fairly self-contained part of the language.

So far we have only considered sequential programs in which statements are obeyed in order. In many applications, it is convenient to write a program as several parallel activities which cooperate as necessary. This is particularly true of programs which interact in real time with physical processes in the real world; simulation programs (which mimic parallel activities in the real world), and programs used to exploit multiprocessor architectures directly.

As a simple example, consider a family going shopping to buy ingredients for a meal. Suppose they need meat, salad and wine and the purchase of these items can be done by calling procedures.

```
procedure SHOPPING is
begin
   BUY_MEAT;
   BUY_SALAD;
   BUY_WINE;
end SHOPPING
```

Screen 103

However, the previous solution corresponds to the family buying each item in sequence. It would be far more efficient for them to split up, so that, for example, mother buys the meat, the children buy the salad and father buys the wine. They agree to meet again, perhaps at the car, when they are done.

```
procedure SHOPPING is
   task GET_SALAD;  --- specification

   task body GET_SALAD is  -- body
   begin
      BUY_SALAD;
   end GET_SALAD;

   task GET_WINE;

   task body GET_WINE is
   begin
      BUY_WINE;
   end GET_WINE;

begin        -- main processor represents
   BUY_MEAT; -- mother and calls BUY_MEAT
end SHOPPING;
```

It is important to realize that the main program is itself considered to be called by a hypothetical main task. We can now trace the sequence of actions when this main task calls the procedure SHOPPING. First, the tasks GET_SALAD and GET_WINE are declared and then, when the main task reaches the 'begin', these dependent tasks are activated in parallel with the main task. The dependent tasks call their respective procedures and terminate. Meanwhile, the main task calls BUY_MEAT, reaches the end of SHOPPING, and waits until all dependent tasks have terminated. (Mom waiting for Dad and kids at the car).

Screen 104

In the SHOPPING example, the various tasks did not interact with each other once they had been set active, except that their parent unit had to wait for them to terminate. Generally, however, tasks will interact with each other during their lifetime. In Ada, this is done by a mechanism known as rendezvous. This is similar to the human situation where two people meet, perform a trans-action and then go on independently.

A rendezvous between two tasks occurs as a consequence of one task calling an entry declared in another. An entry is declared in a task specification in a similar way to a procedure in a package specification. The statements to be obeyed during a rendezvous are described by corresponding accept statements in the body of the task containing the declaration of the entry.

As a more abstract example, consider the problem of providing a task to act as a single buffer between one or more tasks producing items and one or more tasks consuming them. Out intermediate task can hold just one of them.

```
task BUFFERING is
   entry PUT(X:in ITEM);
   entry GET(X:out ITEM);
end BUFFERING;
```

```
task body BUFFERING is
   V            :ITEM;
begin
   loop
     accept PUT(X:in ITEM) do V := X;  end PUT;
     accept GET(X:out ITEM) do X := V;  end GET;
   end loop;
end BUFFERING;
```

Screen 105

SIMPLE SELECT STATEMENTS 12-4

The select statement allows a task to select from one of several rendezvous. Suppose that the type ITEM is a record of multiple values. We want to ensure that the record is not read while it is being written to. This would result in some new values and some old values. The solution is to use a task rather than a package, and entry calls rather than procedure calls. The task will start with an entry WRITE; this ensures that the first call accepted is for WRITE so that there is no risk of a variable being read before it is assigned a value. The task then enters an endless loop containing a single select statement. Each time round the loop, the task will accept a call of READ or WRITE according to the demands upon it. It thus prevents multiple access to the variable V, since it can only deal with one call at a time.

```
task PROTECTED_VARIABLE is
   entry READ(X:out ITEM);
   entry WRITE(X:in ITEM);
end;

task body PROTECTED_VARIABLE is
   V             :ITEM;
begin
   accept WRITE(X:in ITEM) do
      V := X;
   end;
   loop
      select
         accept READ(X:out ITEM) do
            X := V;
         end;
      or
         accept WRITE(X:in ITEM) do
            V := X;
         end;
      end select;
   end loop;
end PROTECTED_VARIABLE;
```

Screen 106

MORE SELECT STATEMENTS 12-5

A more complex form of select statement is illustrated by the classic problem of the bounded buffer. The objective of the task is to allow items to be added to and removed from the buffer in the first-in first-out manner, but to prevent the buffer from being over-filled or under-emptied. This is done with a more general form of select statement which includes the use of guarding conditions. Each branch of the select statement commences with: when (condition) => and is then followed by an accept statement and then some additional statements.

```
task body BUFFERING is
   N         :constant := 8; -- example
   A         :array(1..N) of ITEM;
   COUNT     :INTEGER range 0..N := 0;
begin
   loop
      select
         when COUNT < N =>
            accept PUT(X:in ITEM) do
               A(I) := X;
            end;
            I := I mod N + 1;
            COUNT := COUNT + 1;
      or
         when COUNT > 0 =>
            accept GET(X:out ITEM) do
               X := A(J);X := A(J);
            end;
            J := J mod N + 1;
            COUNT := COUNT - 1;
         end select;
      end loop;
   end BUFFERING;
```

So the guarding conditions are conditions which have to be true before a service can be offered.

Screen 107

If we want to regulate the timing of the tasks, a delay statement can be used. The expression after the word 'delay' is of a predefined fixed point type DURATION and gives the period in seconds.

```
delay 3.0;  -- suspends the task executing the statement
            -- for 3 seconds
```

Delays can be more easily expressed by using suitable constant declarations.

```
SECONDS        :constant DURATION := 1.0;    -- from the package
MINUTES        :constant DURATION := 60.0;   --     CALENDAR
HOURS          :constant DURATION := 3600.0;
```

We can write, for example: `delay 2 * HOURS + 40 * MINUTES;`

Suppose we want to call a procedure ACTION at regular intervals, every five minutes perhaps. Our first attempt might be to write

```
loop
   delay 5 * MINUTES;
   ACTION;
end loop;
```

Screen 108

INTERVALS 12-7

```
   loop
      delay 5 * MINUTES;
      ACTION;
   end loop;
```

This is unsatisfactory for various reasons. First, we have not taken account of the time to execution the procedure ACTION and the overhead of the loop itself, and secondly, a delay statement sets a minimum delay only. Furthermore, we might get preempted by a higher priority task at any time.So we will inevitably get a cumulative timing drift. This can be overcome by

```
   declare
      use CALENDAR;           -- predefined package
      INTERVAL       :constant DURATION := 5 * MINUTES
      NEXT_TIME      :TIME := FIRST_TIME;    -- time when ACTION is called
next
   begin
      loop                                     -- CLOCK returns date and time
         delay NEXT_TIME - CLOCK;              -- ACTION is delayed by the
         ACTION;                               -- difference between NEXT_TIME
         NEXT_TIME := NEXT_TIME + INTERVAL;    -- and the current time
      end loop;
   end;
```

Screen 109

A task can become completed and then terminate in various ways, as well as running into its final end. In an earlier example, the body of a task was an endless loop and clearly never terminated. This means that it would never be possible to leave the unit on which the task was dependent. It is possible to make a task automatically terminate itself when it is of no further use.

```
task body PROTECTED_VARIABLE is
   V                  :ITEM;
begin
   accept WRITE(X:in ITEM) do V := X;
   end;
   loop
      select
         accept READ(X:out ITEM) do
            X := V;
            end;
         or
            accept WRITE(X:in ITEM)
do
               V := X;
            end;
         or
            terminate;
         end select;
      end loop;
   end PROTECTED_VARIABLE;
```

The terminate alternative is taken if the unit on which the task depends has reached its end and so is completed and all sibling tasks and dependent tasks are terminated or are similarly able to select a terminate alternative. In such circumstances, all the tasks are of no use since they are the only tasks that could call their entries and they are all dormant. Thus the whole set automatically terminates.

Screen 110

Selection of a terminate alternative is classified as a normal termination. The task is under control of the situation and terminates voluntarily. At the other extreme is the abort statement. The abort statement unconditionally terminates one or more tasks. It consists of the reserved word 'abort' followed by a list of task names. If a task is aborted, then all tasks dependent upon it or a subprogram or black currently called by it are also aborted.

The abort statement is very disruptive and should only be used in extreme situations. It might be appropriate for a command task to abort a complete subsystem in response to an operator command. Another appropriate use for the abort statement might be an exception handler. When an exception is raised in a unit, we cannot tidy up that unit and propagate the exception on a layered basis while dependent tasks are still alive and so one of the actions of tidying up might be to abort all dependent tasks. However, it is probably always best to attempt a controlled shutdown and only resort to the abort statement as a desperate measure. Statements in the command task might be

```
select
   T.CLOSEDOWN;
or
   delay 60 * SECONDS;
   abort T;
end select;
```

Screen 111

APPENDIX C

```
-- ADA.Y
--
--
-- The following Ada grammar for Ayacc has been adapted
-- by David Taback and Deepak Tolani from the one distributed
-- by Herman Fischer for the use with Yacc.
-- We have made the following modifications to the Grammar.
--
--      1) We have removed the C style comments.
--
--      2) We have added the production
--              numeric_literal
--                      : REAL_LITERAL
--                      | INTEGER_LITERAL
--                      ;
--
--      3) The token names have been converted to uppercase and
--         the name TOKEN has been appendend to them. Some of the
--         names of the other tokens have also been changed.
--
--      4) We replaced '\'' with ''' in the productions.
--
--    5) We added tokens ERROR1 to ERROR15 that was used to interface
--       to an existing Ada lexical analyzer. These tokens are not
--       used in the productions.
--
--      6) We added more 'error' productions to for better error recovery.
--
-- This is the orginal header given with the Yacc version of the
-- grammar.
--

---------------------------------------------------------------------------
--
--                      A LALR(1) grammar for ANSI Ada*
--
--                      Adapted for YACC (UNIX) Inputs
--
--
--                  Herman Fischer
--                Litton Data Systems
--              8000 Woodley Ave., ms 44-30
--                  Van Nuys, CA
--
--                  818/902-5139
--                HFischer@eclb.arpa
--              {cepu,trwrb}!litvax!fischer
--
--                  March 26, 1984
```

```
--
--              A Contribution to the Public Domain
--                            for
--         Research, Development, and Training Purposes Only
--
--     Any Corrections or Problems are to be Reported to the Author
--
--
--
--                      adapted from
--                      the grammar
--                         by:
--
--         Gerry Fisher                     Philippe Charles
--
--   Computer Sciences Corporation    &          Ada  Project
--       4045 Hancock Street               New York University
--       San Diego, CA 92121                251 Mercer Street
--                                 New York, New York 10012
--
--
--   This grammar is organized in the same order as the syntax summary
-- in appendix E of the ANSI Ada Reference Manual.  All reserved words
-- are   written  in  upper  case  letters.   The  lexical  categories
-- numeric_literal, string_literal, etc, are viewed as terminals.  The
-- rules  for  pragmas as  stated in  chapter 2,  section 8,  have been
-- incorporated in the grammar.  Comments are included wherever we had
-- to deviate from the syntax given in appendix E. Different symbols
-- used here (to comply with yacc requirements) are of note:
--        {,something}  is denoted ...something..
--        {something}   is denoted ..something..
--        [something]   is denoted .something.
-- Constructs involving
-- meta brackets, e.g., ...identifier.. are represented by a nonterminal
-- formed  by  concatenating  the  construct symbols ( as ...identifier..
-- in the example)   for  which  the  rules are given at the end.  When
-- reading  this  grammar,  it   is  important to  note that all symbols
-- appearing  in the  rules are  separated by  one or  more  blanks.  A
-- string  such  as  'identifier_type_mark  is  actually  a  single
-- nonterminal symbol defined at the end of the rules.  The '/*' symbol
-- is used  to indicate that the rest of  the  line is a comment,  just
-- as in yacc programs.
--
--    This grammar is presented here in a form suitable for input to a
-- yacc parser generator.  It has  been  processed  by the Bell System
-- III lex/yacc combination, and tested against over 400 ACVC tests.
--
-- *Ada is a registered trade mark of the  Department of  Defense  (Ada
-- Joint Program  Office).
--
--------------------------------------------------------------------------
```

```
%token '&' ''' '(' ')' '*' '+' ',' '-' '.' '/' ':' ';'
%token '<' '=' '>' '|'

%token ARROW DOUBLE_DOT DOUBLE_STAR ASSIGNMENT INEQUALITY
%token GREATER_THAN_OR_EQUAL LESS_THAN_OR_EQUAL
%token LEFT_LABEL_BRACKET RIGHT_LABEL_BRACKET
%token BOX

%token ABORT_TOKEN ABS_TOKEN ACCEPT_TOKEN  ACCESS_TOKEN
%token ALL_TOKEN  AND_TOKEN ARRAY_TOKEN AT_TOKEN

%token BEGIN_TOKEN  BODY_TOKEN

%token CASE_TOKEN  CONSTANT_TOKEN

%token DECLARE_TOKEN DELAY_TOKEN DELTA_TOKEN DIGITS_TOKEN DO_TOKEN

%token ELSE_TOKEN ELSIF_TOKEN END_TOKEN  ENTRY_TOKEN EXCEPTION_TOKEN
%token EXIT_TOKEN

%token FOR_TOKEN FUNCTION_TOKEN

%token GENERIC_TOKEN  GOTO_TOKEN

%token IF_TOKEN IN_TOKEN  IS_TOKEN

%token LIMITED_TOKEN LOOP_TOKEN

%token MOD_TOKEN

%token NEW_TOKEN NOT_TOKEN  NULL_TOKEN

%token OF_TOKEN OR_TOKEN OTHERS_TOKEN OUT_TOKEN

%token PACKAGE_TOKEN PRAGMA_TOKEN  PRIVATE_TOKEN PROCEDURE_TOKEN

%token RAISE_TOKEN  RANGE_TOKEN RECORD_TOKEN REM_TOKEN RENAMES_TOKEN
%token RETURN_TOKEN REVERSE_TOKEN

%token SELECT_TOKEN  SEPARATE_TOKEN SUBTYPE_TOKEN

%token TASK_TOKEN TERMINATE_TOKEN THEN_TOKEN TYPE_TOKEN

%token USE_TOKEN

%token WHEN_TOKEN WHILE_TOKEN WITH_TOKEN

%token XOR_TOKEN

%token IDENTIFIER
%token INTEGER_LITERAL REAL_LITERAL
```

```
%token CHARACTER_LITERAL STRING_LITERAL

%token ERROR1 ERROR2  ERROR3  ERROR4  ERROR5  ERROR6  ERROR7  ERROR8
%token ERROR9 ERROR10 ERROR11 ERROR12 ERROR13 ERROR14 ERROR15

%start compilation

{
  subtype yystype is integer;
}


%%

prag    :
        PRAGMA_TOKEN IDENTIFIER .arg_ascs ';'
        {assign_association(yy.rule_id);} ;

arg_asc                 :
        expr
  |     IDENTIFIER ARROW expr  {assign_association(yy.rule_id);} ;

 ***   Added    *** --
numeric_literal
        : REAL_LITERAL
        | INTEGER_LITERAL
        {assign_association(yy.rule_id);} ;

basic_d  :
        object_d
  |     ty_d          | subty_d
  |     subprg_d       | pkg_d
  |     task_d        | gen_d
  |     excptn_d       | gen_inst
  |   renaming_d    | number_d
  |    error ';' {assign_association(yy.rule_id);} ;

object_d  :
    idents ':'          subty_ind ._ASN_expr. ';'
 |  idents ':'  CONSTANT_TOKEN subty_ind ._ASN_expr. ';'
 |  idents ':'          c_arr_def ._ASN_expr. ';'
 |  idents ':'  CONSTANT_TOKEN c_arr_def ._ASN_expr. ';'
        {assign_association(yy.rule_id);} ;

number_d  :
        idents ':' CONSTANT_TOKEN ASSIGNMENT expr ';'
        {assign_association(yy.rule_id);} ;

idents   : IDENTIFIER ...ident..
        {assign_association(yy.rule_id);} ;

ty_d  :
```

```
        full_ty_d
  |     incomplete_ty_d
  |     priv_ty_d  {assign_association(yy.rule_id);} ;

full_ty_d :
      TYPE_TOKEN IDENTIFIER            IS_TOKEN ty_def ';'
  |   TYPE_TOKEN IDENTIFIER discr_part IS_TOKEN ty_def ';'
        {assign_association(yy.rule_id);} ;

ty_def  :
        enum_ty_def             | integer_ty_def
  |     real_ty_def                     | array_ty_def
  |     rec_ty_def              | access_ty_def
  |     derived_ty_def {assign_association(yy.rule_id);} ;

subty_d  :
        SUBTYPE_TOKEN IDENTIFIER IS_TOKEN subty_ind ';'
        {assign_association(yy.rule_id);} ;

subty_ind  :    ty_mk .constrt.
        {assign_association(yy.rule_id);} ;

ty_mk  : expanded_n {assign_association(yy.rule_id);} ;

constrt  :
        rng_c
  |     fltg_point_c | fixed_point_c
  |     aggr {assign_association(yy.rule_id);} ;

derived_ty_def            :  NEW_TOKEN subty_ind
        {assign_association(yy.rule_id);} ;

rng_c  :  RANGE_TOKEN rng {assign_association(yy.rule_id);} ;

rng  :
        name
  |     sim_expr DOUBLE_DOT sim_expr
        {assign_association(yy.rule_id);} ;

enum_ty_def        :
        '(' enum_lit_spec
                        ...enum_lit_spec..  ')'
        {assign_association(yy.rule_id);} ;

enum_lit_spec  :  enum_lit {assign_association(yy.rule_id);} ;

enum_lit  : IDENTIFIER | CHARACTER_LITERAL
        {assign_association(yy.rule_id);} ;

integer_ty_def            :  rng_c {assign_association(yy.rule_id);} ;

real_ty_def          :
```

```
        fltg_point_c | fixed_point_c {assign_association(yy.rule_id);} ;

fltg_point_c :
        fltg_accuracy_def .rng_c. {assign_association(yy.rule_id);} ;

fltg_accuracy_def       :
        DIGITS_TOKEN sim_expr {assign_association(yy.rule_id);} ;

fixed_point_c :
        fixed_accuracy_def .rng_c. {assign_association(yy.rule_id);} ;

fixed_accuracy_def      :
        DELTA_TOKEN sim_expr {assign_association(yy.rule_id);} ;

array_ty_def        :
        uncnstrnd_array_def           | c_arr_def
        {assign_association(yy.rule_id);} ;

uncnstrnd_array_def          :
      ARRAY_TOKEN '(' idx_subty_def ...idx_subty_def.. ')' OF_TOKEN
                                               subty_ind
        {assign_association(yy.rule_id);} ;

c_arr_def          :
        ARRAY_TOKEN idx_c OF_TOKEN subty_ind
        {assign_association(yy.rule_id);} ;

idx_subty_def    :    name  RANGE_TOKEN BOX
        {assign_association(yy.rule_id);} ;

idx_c  :  '(' dscr_rng ...dscr_rng.. ')' {assign_association(yy.rule_id);} ;

dscr_rng        :
        rng
  |     name rng_c     {assign_association(yy.rule_id);} ;

rec_ty_def      :
        RECORD_TOKEN
           cmpons
        END_TOKEN RECORD_TOKEN {assign_association(yy.rule_id);} ;

cmpons          :
        ..prag.. ..cmpon_d.. cmpon_d ..prag..
  |     ..prag.. ..cmpon_d.. variant_part ..prag..
   |   ..prag.. NULL_TOKEN ';' ..prag..
        {assign_association(yy.rule_id);} ;

cmpon_d :
    idents ':' cmpon_subty_def ._ASN_expr. ';'
        {assign_association(yy.rule_id);} ;

cmpon_subty_def          : subty_ind {assign_association(yy.rule_id);} ;
```

```
discr_part :
        '(' discr_spec ...discr_spec.. ')'
          {assign_association(yy.rule_id);} ;

discr_spec :
          idents ':' ty_mk ._ASN_expr. {assign_association(yy.rule_id);} ;

variant_part :
          CASE_TOKEN sim_n IS_TOKEN
                ..prag.. variant ..variant..
       END_TOKEN CASE_TOKEN ';' {assign_association(yy.rule_id);} ;

variant :
          WHEN_TOKEN choice ..or_choice.. ARROW
             cmpons {assign_association(yy.rule_id);} ;

choice : sim_expr
          | name rng_c
          | sim_expr DOUBLE_DOT sim_expr
          | OTHERS_TOKEN
          | error
          {assign_association(yy.rule_id);} ;

access_ty_def          : ACCESS_TOKEN subty_ind {assign_association(yy.rule_id);} ;

incomplete_ty_d :
       TYPE_TOKEN IDENTIFIER ';'
  |    TYPE_TOKEN IDENTIFIER discr_part ';'
          {assign_association(yy.rule_id);} ;

decl_part          :
          ..basic_decl_item..
  |    ..basic_decl_item.. body ..later_decl_item..
          {assign_association(yy.rule_id);} ;

basic_decl_item        :
          basic_d
  |       rep_cl | use_cl         {assign_association(yy.rule_id);} ;

later_decl_item        : body
  |       subprg_d       | pkg_d
  |       task_d | gen_d
  |       use_cl         | gen_inst
          {assign_association(yy.rule_id);} ;

body : proper_body | body_stub {assign_association(yy.rule_id);} ;

proper_body :
          subprg_body | pkg_body | task_body
          {assign_association(yy.rule_id);} ;
```

```
name : sim_n
  | CHARACTER_LITERAL | op_symbol
  |     idxed_cmpon
  |     selected_cmpon | attribute {assign_association(yy.rule_id);} ;

sim_n :         IDENTIFIER {assign_association(yy.rule_id);} ;

prefix : name {assign_association(yy.rule_id);} ;

idxed_cmpon :
        prefix aggr {assign_association(yy.rule_id);} ;

selected_cmpon : prefix '.' selector ;

selector : sim_n
  | CHARACTER_LITERAL | op_symbol | ALL_TOKEN
        {assign_association(yy.rule_id);} ;

attribute : prefix ''' attribute_designator
        {assign_association(yy.rule_id);} ;

attribute_designator :
        sim_n
  |     DIGITS_TOKEN
  |     DELTA_TOKEN
  |     RANGE_TOKEN       {assign_association(yy.rule_id);} ;

aggr :
      '(' cmpon_asc ...cmpon_asc.. ')' {assign_association(yy.rule_id);} ;

cmpon_asc      :
        expr
  |     choice ..or_choice.. ARROW expr
  |     sim_expr DOUBLE_DOT sim_expr
  |     name rng_c {assign_association(yy.rule_id);} ;

expr :
        rel..AND__rel.. | rel..AND__THEN__rel..
  |     rel..OR__rel.. | rel..OR__ELSE__rel..
  |     rel..XOR__rel.. {assign_association(yy.rule_id);} ;

rel :
        sim_expr .relal_op__sim_expr.
  |     sim_expr.NOT.IN__rng_or_sim_expr.NOT.IN__ty_mk
        {assign_association(yy.rule_id);} ;

sim_expr :
        .unary_add_op.term..binary_add_op__term..
        {assign_association(yy.rule_id);} ;

term : factor..mult_op__factor.. {assign_association(yy.rule_id);} ;
```

```
factor  : pri ._EXP___pri. | ABS_TOKEN pri |   NOT_TOKEN pri
        {assign_association(yy.rule_id);} ;

pri :
        numeric_literal  | NULL_TOKEN
  |     allocator | qualified_expr
  |     name
  |     aggr {assign_association(yy.rule_id);} ;

relal_op  : '='
          | INEQUALITY
          | '<'
          | LESS_THAN_OR_EQUAL
          | '>'
          | GREATER_THAN_OR_EQUAL  {assign_association(yy.rule_id);} ;

binary_add_op  : '+' | '-' | '&'
        {assign_association(yy.rule_id);} ;

unary_add_op  : '+' | '-'  {assign_association(yy.rule_id);} ;

mult_op  : '*' | '/' | MOD_TOKEN | REM_TOKEN
        {assign_association(yy.rule_id);} ;

qualified_expr          :
        ty_mkaggr_or_ty_mkPexprP_ {assign_association(yy.rule_id);} ;

allocator  :
        NEW_TOKEN ty_mk
  |     NEW_TOKEN ty_mk aggr
  |  NEW_TOKEN ty_mk ''' aggr  {assign_association(yy.rule_id);} ;

seq_of_stmts  : ..prag.. stmt ..stmt.. { null; } -- Because of bug
              {assign_association(yy.rule_id);} ;

stmt  :
        ..label.. sim_stmt | ..label.. compound_stmt
  |  error   ';' {assign_association(yy.rule_id);} ;

sim_stmt :      null_stmt
  |     assignment_stmt  | exit_stmt
  |     return_stmt      | goto_stmt
  |     delay_stmt       | abort_stmt
  |     raise_stmt       | code_stmt
  |  name ';' {assign_association(yy.rule_id);} ;

compound_stmt  :
        if_stmt  | case_stmt
  |     loop_stmt        | block_stmt
  |     accept_stmt |  select_stmt  {assign_association(yy.rule_id);} ;

label  : LEFT_LABEL_BRACKET sim_n RIGHT_LABEL_BRACKET
```

```
        {assign_association(yy.rule_id);} ;

null_stmt  :  NULL_TOKEN ';'  {assign_association(yy.rule_id);} ;

assignment_stmt  :
      name ASSIGNMENT expr ';'
        {assign_association(yy.rule_id);} ;

if_stmt  :
        IF_TOKEN cond THEN_TOKEN
           seq_of_stmts
        ..ELSIF__cond__THEN__seq_of_stmts..
        .ELSE__seq_of_stmts.
     END_TOKEN IF_TOKEN ';' {assign_association(yy.rule_id);} ;

cond  :  expr {assign_association(yy.rule_id);} ;

case_stmt        :
        CASE_TOKEN expr IS_TOKEN
           case_stmt_alt..case_stmt_alt..
     END_TOKEN CASE_TOKEN ';'
  {assign_association(yy.rule_id);} ;

case_stmt_alt  :
        WHEN_TOKEN choice ..or_choice.. ARROW
           seq_of_stmts {assign_association(yy.rule_id);} ;

loop_stmt        :
        .sim_nC.
           .iteration_scheme. LOOP_TOKEN
                 seq_of_stmts
        END_TOKEN LOOP_TOKEN .sim_n. ';'
        {assign_association(yy.rule_id);} ;

iteration_scheme
        : WHILE_TOKEN cond
        | WHILE_TOKEN error
        | FOR_TOKEN loop_prm_spec
        | FOR_TOKEN error
        {assign_association(yy.rule_id);} ;

loop_prm_spec  :
        IDENTIFIER IN_TOKEN .REVERSE. dscr_rng
        {assign_association(yy.rule_id);} ;

block_stmt  :
        .sim_nC.
           .DECLARE__decl_part.
           BEGIN_TOKEN
                 seq_of_stmts
           .EXCEPTION__excptn_handler..excptn_handler...
        END_TOKEN .sim_n. ';' {assign_association(yy.rule_id);} ;
```

```
exit_stmt        :
      EXIT_TOKEN .expanded_n. .WHEN__cond. ';'
        {assign_association(yy.rule_id);} ;

return_stmt  :  RETURN_TOKEN .expr. ';' {assign_association(yy.rule_id);} ;

goto_stmt  :  GOTO_TOKEN expanded_n ';' {assign_association(yy.rule_id);} ;

subprg_d  :  subprg_spec ';'  {assign_association(yy.rule_id);} ;

subprg_spec  :
        PROCEDURE_TOKEN IDENTIFIER .fml_part.
  |     FUNCTION_TOKEN  designator .fml_part. RETURN_TOKEN ty_mk
        {assign_association(yy.rule_id);} ;

designator  :  IDENTIFIER  |  op_symbol  {assign_association(yy.rule_id);} ;

op_symbol  :  STRING_LITERAL  {assign_association(yy.rule_id);} ;

fml_part  :
      '(' prm_spec .._.prm_spec.. ')' {assign_association(yy.rule_id);} ;

prm_spec  :
        idents ':' mode ty_mk ._ASN_expr.  {assign_association(yy.rule_id);} ;

mode  :  .IN.  |  IN_TOKEN OUT_TOKEN  |  OUT_TOKEN
        {assign_association(yy.rule_id);} ;

subprg_body  :
        subprg_spec IS_TOKEN
           .decl_part.
        BEGIN_TOKEN
           seq_of_stmts
        .EXCEPTION__excptn_handler..excptn_handler...
     END_TOKEN .designator. ';' {assign_association(yy.rule_id);} ;

pkg_d  :  pkg_spec ';' {assign_association(yy.rule_id);}  ;

pkg_spec  :
        PACKAGE_TOKEN IDENTIFIER IS_TOKEN
           ..basic_decl_item..
        .PRIVATE..basic_decl_item...
        END_TOKEN .sim_n.   {assign_association(yy.rule_id);} ;

pkg_body  :
        PACKAGE_TOKEN BODY_TOKEN sim_n IS_TOKEN
           .decl_part.
           .BEGIN__seq_of_stmts.EXCEPTION__excptn_handler..excptn_handler...
        END_TOKEN .sim_n. ';'   {assign_association(yy.rule_id);} ;

priv_ty_d  :
```

```
        TYPE_TOKEN IDENTIFIER           IS_TOKEN .LIMITED. PRIVATE_TOKEN ';'
    | TYPE_TOKEN IDENTIFIER discr_part IS_TOKEN .LIMITED. PRIVATE_TOKEN ';'
        {assign_association(yy.rule_id);} ;

use_cl  :  USE_TOKEN expanded_n ...expanded_n.. ';'
        {assign_association(yy.rule_id);} ;

renaming_d  :
      idents ':' ty_mk       RENAMES_TOKEN name ';'
  |   idents ':' EXCEPTION_TOKEN      RENAMES_TOKEN expanded_n ';'
  |   PACKAGE_TOKEN IDENTIFIER        RENAMES_TOKEN expanded_n ';'
  |   subprg_spec  RENAMES_TOKEN name ';'
        {assign_association(yy.rule_id);} ;

task_d  :  task_spec ';'  {assign_association(yy.rule_id);} ;

task_spec  :
        TASK_TOKEN .TYPE. IDENTIFIER
          .IS..ent_d_..rep_cl_END.sim_n.
          {assign_association(yy.rule_id);} ;

task_body  :
        TASK_TOKEN BODY_TOKEN sim_n IS_TOKEN
          .decl_part.
        BEGIN_TOKEN
          seq_of_stmts
        .EXCEPTION__excptn_handler..excptn_handler...
      END_TOKEN .sim_n. ';' {assign_association(yy.rule_id);} ;

ent_d  :
        ENTRY_TOKEN IDENTIFIER .fml_part. ';'
  |   ENTRY_TOKEN IDENTIFIER '(' dscr_rng ')' .fml_part. ';'
        {assign_association(yy.rule_id);} ;

ent_call_stmt  :
      ..prag.. name ';' {assign_association(yy.rule_id);} ;

accept_stmt  :
        ACCEPT_TOKEN sim_n .Pent_idx_P..fml_part.
        .DO__seq_of_stmts__END.sim_n.. ';'
        {assign_association(yy.rule_id);} ;

ent_idx  :        expr   {assign_association(yy.rule_id);} ;

delay_stmt  :  DELAY_TOKEN sim_expr ';' {assign_association(yy.rule_id);} ;

select_stmt  :   selec_wait
  |       condal_ent_call          | timed_ent_call  {assign_association(yy.rule_id);} ;

selec_wait      :
        SELECT_TOKEN
          select_alt
```

```
            ..OR__select_alt..
            .ELSE__seq_of_stmts.
      END_TOKEN SELECT_TOKEN ';' {assign_association(yy.rule_id);} ;

select_alt :
         .WHEN__condARROW.selec_wait_alt {assign_association(yy.rule_id);} ;

selec_wait_alt : accept_alt
  |      delay_alt | terminate_alt {assign_association(yy.rule_id);} ;

accept_alt :
         accept_stmt.seq_of_stmts. {assign_association(yy.rule_id);} ;

delay_alt :
         delay_stmt.seq_of_stmts. {assign_association(yy.rule_id);} ;

terminate_alt : TERM_stmt  {assign_association(yy.rule_id);} ;

condal_ent_call          :
         SELECT_TOKEN
           ent_call_stmt
            .seq_of_stmts.
         ELSE_TOKEN
            seq_of_stmts
      END_TOKEN SELECT_TOKEN ';' {assign_association(yy.rule_id);} ;

timed_ent_call :
         SELECT_TOKEN
           ent_call_stmt
            .seq_of_stmts.
         OR_TOKEN
           delay_alt
      END_TOKEN SELECT_TOKEN ';' {assign_association(yy.rule_id);} ;

abort_stmt : ABORT_TOKEN name ...name.. ';'
         {assign_association(yy.rule_id);} ;

compilation :  ..compilation_unit..  {assign_association(yy.rule_id);}  ;

compilation_unit :
         context_cl library_unit
  |   context_cl secondary_unit {assign_association(yy.rule_id);} ;

library_unit :
         subprg_d       | pkg_d
  |      gen_d | gen_inst
  |      subprg_body    {assign_association(yy.rule_id);} ;

secondary_unit           :
         library_unit_body | subunit   {assign_association(yy.rule_id);} ;

library_unit_body :
```

```
        pkg_body_or_subprg_body  {assign_association(yy.rule_id);} ;

context_cl      : ..with_cl..use_cl.... {assign_association(yy.rule_id);} ;

with_cl  :  WITH_TOKEN sim_n ...sim_n.. ';'
        {assign_association(yy.rule_id);} ;

body_stub  :
      subprg_spec IS_TOKEN SEPARATE_TOKEN ';'
  |   PACKAGE_TOKEN BODY_TOKEN sim_n IS_TOKEN SEPARATE_TOKEN ';'
  |   TASK_TOKEN BODY_TOKEN sim_n    IS_TOKEN SEPARATE_TOKEN ';'
        {assign_association(yy.rule_id);} ;

subunit  :  SEPARATE_TOKEN '(' expanded_n ')' proper_body
        {assign_association(yy.rule_id);} ;

excptn_d  :  idents ':' EXCEPTION_TOKEN ';'
        {assign_association(yy.rule_id);} ;

excptn_handler          :
        WHEN_TOKEN excptn_choice ..or_excptn_choice.. ARROW
          seq_of_stmts  {assign_association(yy.rule_id);} ;

excptn_choice  :  expanded_n  |         OTHERS_TOKEN
        {assign_association(yy.rule_id);} ;

raise_stmt  :  RAISE_TOKEN .expanded_n. ';' {assign_association(yy.rule_id);} ;

gen_d  :  gen_spec ';'  {assign_association(yy.rule_id);} ;

gen_spec  :
        gen_fml_part subprg_spec
  |     gen_fml_part pkg_spec  {assign_association(yy.rule_id);} ;

gen_fml_part  :          GENERIC_TOKEN ..gen_prm_d..
        {assign_association(yy.rule_id);} ;

gen_prm_d  :
        idents ':' .IN.OUT.. ty_mk ._ASN_expr. ';'
  |   TYPE_TOKEN IDENTIFIER IS_TOKEN gen_ty_def ';'
  |     priv_ty_d
  |   WITH_TOKEN subprg_spec .IS_BOX_. ';'
        {assign_association(yy.rule_id);} ;

gen_ty_def      :
      '(' BOX ')'  |  RANGE_TOKEN BOX  |  DIGITS_TOKEN BOX  |  DELTA_TOKEN BOX
  |     array_ty_def        | access_ty_def
        {assign_association(yy.rule_id);} ;

gen_inst  :
        PACKAGE_TOKEN IDENTIFIER IS_TOKEN
        NEW_TOKEN expanded_n .gen_act_part. ';'
```

```
 |      PROCEDURE__ident__IS_
        NEW_TOKEN expanded_n .gen_act_part. ';'
 |      FUNCTION_TOKEN  designator IS_TOKEN
           NEW_TOKEN expanded_n .gen_act_part. ';'
        {assign_association(yy.rule_id);} ;

gen_act_part  :
       '(' gen_asc ...gen_asc.. ')'
        {assign_association(yy.rule_id);} ;

gen_asc              :
        .gen_fml_prmARROW.gen_act_prm {assign_association(yy.rule_id);} ;

gen_fml_prm  :
        sim_n  |  op_symbol {assign_association(yy.rule_id);}  ;

gen_act_prm  :
  expr_or_name_or_subprg_n_or_ent_n_or_ty_mk
         {assign_association(yy.rule_id);} ;

rep_cl  :
        ty_rep_cl  |  address_cl  {assign_association(yy.rule_id);} ;

ty_rep_cl  :  length_cl
 |      enum_rep_cl
 |      rec_rep_cl          {assign_association(yy.rule_id);} ;

length_cl  :  FOR_TOKEN attribute USE_TOKEN sim_expr ';'
        {assign_association(yy.rule_id);} ;

enum_rep_cl  :
      FOR__ty_sim_n__USE_ aggr ';' {assign_association(yy.rule_id);} ;

rec_rep_cl       :
        FOR__ty_sim_n__USE_
           RECORD_TOKEN .algt_cl.
                ..cmpon_cl..
        END_TOKEN RECORD_TOKEN ';'    {assign_association(yy.rule_id);} ;

algt_cl  :     AT_TOKEN MOD_TOKEN sim_expr ';'
        {assign_association(yy.rule_id);} ;

cmpon_cl  :
      name AT_TOKEN sim_expr RANGE_TOKEN rng ';'
        {assign_association(yy.rule_id);} ;

address_cl  :  FOR_TOKEN sim_n USE_TOKEN AT_TOKEN sim_expr ';'
        {assign_association(yy.rule_id);} ;

code_stmt  :  ty_mk_rec_aggr ';' {assign_association(yy.rule_id);} ;

..prag..  :
```

```
 |      ..prag.. prag  {assign_association(yy.rule_id);} ;

.arg_ascs :
 |   '(' arg_ascs ')' {assign_association(yy.rule_id);} ;

arg_ascs :
        arg_asc
 |   arg_ascs ',' arg_asc {assign_association(yy.rule_id);} ;

._ASN_expr.  :
 |      ASSIGNMENT expr  {assign_association(yy.rule_id);} ;

...ident    :
 |   ...[illegible]ent.. ',' IDENTIFIER    {assign_association(yy.rule_id);} ;

.constrt.  :
 |      constrt   {assign_association(yy.rule_id);} ;

expanded_n  :
        IDENTIFIER
 |   expanded_n '.' IDENTIFIER    {assign_association(yy.rule_id);} ;

...enum_lit_spec.. :
 |   ...enum_lit_spec.. ','
              enum_lit_spec {assign_association(yy.rule_id);} ;

.rng_c.  :
 |      rng_c   {assign_association(yy.rule_id);} ;

...idx_subty_def..         :
 |   ...idx_subty_def.. ',' idx_subty_def
        {assign_association(yy.rule_id);} ;

...dscr_rng.. :
 |   ...dscr_rng.. ',' dscr_rng  {assign_association(yy.rule_id);} ;

..cmpon_d.. :
 |   ..cmpon_d.. cmpon_d ..prag..    {assign_association(yy.rule_id);} ;

...discr_spec..          :
 |   ...discr_spec.. ';' discr_spec  {assign_association(yy.rule_id);} ;

..variant..  :
 |      ..variant.. variant  {assign_association(yy.rule_id);} ;

..or_choice.. :
 |   ..or_choice.. '|' choice  {assign_association(yy.rule_id);} ;

..basic_decl_item..        :
        ..prag..
 |      ..basic_decl_item.. basic_decl_item ..prag..
        {assign_association(yy.rule_id);} ;
```

```
..later_decl_item..      :
        ..prag..
  |   ..later_decl_item.. later_decl_item ..prag..
        {assign_association(yy.rule_id);} ;

...cmpon_asc..           :
  |   ...cmpon_asc.. ',' cmpon_asc {assign_association(yy.rule_id);} ;

rel..AND__rel..  :
        rel AND_TOKEN rel
  |     rel..AND__rel.. AND_TOKEN rel    {assign_association(yy.rule_id);} ;

rel..OR__rel.. :
        rel OR_TOKEN rel
  |     rel..OR__rel.. OR_TOKEN rel {assign_association(yy.rule_id);} ;

rel..XOR__rel..  :
        rel
  |     ..XOR__rel..  {assign_association(yy.rule_id);} ;

..XOR__rel..  :
        rel XOR_TOKEN rel
  |     ..XOR__rel.. XOR_TOKEN rel      {assign_association(yy.rule_id);} ;

rel..AND__THEN__rel..  :
        rel AND_TOKEN THEN_TOKEN rel
  |     rel..AND__THEN__rel.. AND_TOKEN THEN_TOKEN rel
        {assign_association(yy.rule_id);} ;

rel..OR__ELSE__rel..          :
        rel OR_TOKEN ELSE_TOKEN rel
  |     rel..OR__ELSE__rel.. OR_TOKEN ELSE_TOKEN rel
        {assign_association(yy.rule_id);} ;

.relal_op__sim_expr.  :
  |     relal_op sim_expr  {assign_association(yy.rule_id);} ;

sim_expr.NOT.IN__rng_or_sim_expr.NOT.IN__ty_mk      :
        sim_expr .NOT. IN_TOKEN rng {assign_association(yy.rule_id);} ;

.NOT.  :
  |     NOT_TOKEN    {assign_association(yy.rule_id);} ;

.unary_add_op.term..binary_add_op__term..  :
        term
  |     unary_add_op term
  |     .unary_add_op.term..binary_add_op__term..
        binary_add_op term {assign_association(yy.rule_id);}  ;

factor..mult_op__factor..       :
        factor
```

```
 |      factor..mult_op__factor.. mult_op factor
        {assign_association(yy.rule_id);} ;

._EXP___pri.  :
 |      DOUBLE_STAR pri {assign_association(yy.rule_id);} ;

ty_mkaggr_or_ty_mkPexprP_  :
      prefix "' aggr {assign_association(yy.rule_id);} ;

..stmt..  :
        ..prag..
 |      ..stmt.. stmt ..prag.. {assign_association(yy.rule_id);} ;

..label..  :
 |      ..label.. label {assign_association(yy.rule_id);} ;

..ELSIF__cond__THEN__seq_of_stmts..  :
 |      ..ELSIF__cond__THEN__seq_of_stmts..
                ELSIF_TOKEN cond THEN_TOKEN
                  seq_of_stmts   {assign_association(yy.rule_id);} ;

.ELSE__seq_of_stmts.        :
 |      ELSE_TOKEN
          seq_of_stmts   {assign_association(yy.rule_id);} ;

case_stmt_alt..case_stmt_alt.. :
           ..prag..
          case_stmt_alt
          ..case_stmt_alt..  {assign_association(yy.rule_id);} ;

..case_stmt_alt..        :
 |      ..case_stmt_alt.. case_stmt_alt {assign_association(yy.rule_id);} ;

.sim_nC.       :
 |   sim_n ':' {assign_association(yy.rule_id);} ;

.sim_n. :
 |      sim_n  {assign_association(yy.rule_id);} ;

.iteration_scheme.  :
 |      iteration_scheme  {assign_association(yy.rule_id);} ;

.REVERSE. :
 |      REVERSE_TOKEN   {assign_association(yy.rule_id);} ;

.DECLARE__decl_part.           :
 |      DECLARE_TOKEN
          decl_part              {assign_association(yy.rule_id);} ;

.EXCEPTION__excptn_handler..excptn_handler... :
 |      EXCEPTION_TOKEN
          ..prag.. excptn_handlers     {assign_association(yy.rule_id);} ;
```

```
excptn_handlers          :
        excptn_handler
  |     excptn_handlers excptn_handler   {assign_association(yy.rule_id);} ;

.expanded_n.  :
  |     expanded_n  {assign_association(yy.rule_id);} ;

.WHEN__cond.  :
  |     WHEN_TOKEN cond           {assign_association(yy.rule_id);} ;

.expr.  :
  |     expr   {assign_association(yy.rule_id);} ;

.fml_part.  :
  |     fml_part  {assign_association(yy.rule_id);} ;

.._.prm_spec..  :
  |   .._.prm_spec.. ';' prm_spec  {assign_association(yy.rule_id);} ;

.IN.  :
  |     IN_TOKEN     ;

.decl_part.          :  decl_part   {assign_association(yy.rule_id);} ;

.designator.  :
  |  designator   {assign_association(yy.rule_id);} ;

.PRIVATE..basic_decl_item... :
  |     PRIVATE_TOKEN
           ..basic_decl_item.. {assign_association(yy.rule_id);} ;

.BEGIN__seq_of_stmts.EXCEPTION__excptn_handler..excptn_handler...
                                                :
  |     BEGIN_TOKEN
           seq_of_stmts
        .EXCEPTION__excptn_handler..excptn_handler...
        {assign_association(yy.rule_id);}  ;

.LIMITED.  :
  |     LIMITED_TOKEN    {assign_association(yy.rule_id);} ;

...expanded_n.. :
  |   ...expanded_n.. ',' expanded_n  {assign_association(yy.rule_id);} ;

.TYPE.          :
  |     TYPE_TOKEN   {assign_association(yy.rule_id);} ;

.IS..ent_d_..rep_cl_END.sim_n.  :
  |     IS_TOKEN
            ..ent_d..
            ..rep_cl..
```

```
        END_TOKEN .sim_n.               {assign_association(yy.rule_id);} ;

..ent_d..  :
        ..prag..
  |     ..ent_d.. ent_d ..prag..        {assign_association(yy.rule_id);} ;

..rep_cl..  :
  |     ..rep_cl.. rep_cl ..prag..      {assign_association(yy.rule_id);} ;

.Pent_idx_P..fml_part.  :
        .fml_part.
  |  '(' ent_idx ')' .fml_part.  {assign_association(yy.rule_id);} ;

.DO__seq_of_stmts__END.sim_n..  :
  |     DO_TOKEN
         seq_of_stmts
        END_TOKEN .sim_n.  {assign_association(yy.rule_id);} ;

..OR__select_alt..  :
  |     ..OR__select_alt.. OR_TOKEN select_alt
        {assign_association(yy.rule_id);} ;

.WHEN__condARROW.selec_wait_alt  :
        selec_wait_alt
  |     WHEN_TOKEN cond ARROW selec_wait_alt
        {assign_association(yy.rule_id);} ;

accept_stmt.seq_of_stmts.  :
        ..prag.. accept_stmt .seq_of_stmts.
        {assign_association(yy.rule_id);} ;

delay_stmt.seq_of_stmts.  :
        ..prag.. delay_stmt .seq_of_stmts.
        {assign_association(yy.rule_id);} ;

TERM_stmt   :  ..prag.. TERMINATE_TOKEN ';' ..prag..
        {assign_association(yy.rule_id);} ;

.seq_of_stmts.          :
        ..prag..
  |     seq_of_stmts   {assign_association(yy.rule_id);} ;

...name..       :
  |  ...name.. ',' name  {assign_association(yy.rule_id);} ;

..compilation_unit..  :
        ..prag..
  |     ..compilation_unit.. compilation_unit ..prag..
        {assign_association(yy.rule_id);} ;

pkg_body_or_subprg_body   :  pkg_body
        {assign_association(yy.rule_id);} ;
```

```
..with_cl..use_cl.... :
  |     ..with_cl..use_cl.... with_cl use_cls
        {assign_association(yy.rule_id);} ;

use_cls :
        ..prag..
  |     use_cls use_cl ..prag..
        {assign_association(yy.rule_id);} ;

...sim_n..  :
  |   ...sim_n.. ',' sim_n  {assign_association(yy.rule_id);} ;

..or_excptn_choice.. :
  |   ..or_excptn_choice.. '|' excptn_choice
        {assign_association(yy.rule_id);} ;

..gen_prm_d..  :
  |     ..gen_prm_d.. gen_prm_d {assign_association(yy.rule_id);} ;

.IN.OUT..  :
        .IN.
  |     IN_TOKEN OUT_TOKEN     {assign_association(yy.rule_id);} ;

.IS_BOX_.    :
  |     IS_TOKEN name
  |     IS_TOKEN BOX ;

PROCEDURE__ident__IS_     : subprg_spec IS_TOKEN
        {assign_association(yy.rule_id);} ;

.gen_act_part.  :
  |     gen_act_part {assign_association(yy.rule_id);} ;

...gen_asc.. :
  |   ...gen_asc.. ',' gen_asc {assign_association(yy.rule_id);} ;

.gen_fml_prmARROW.gen_act_prm  :
        gen_act_prm
  |     gen_fml_prm ARROW gen_act_prm  {assign_association(yy.rule_id);} ;

expr_or_name_or_subprg_n_or_ent_n_or_ty_mk
        : expr  {assign_association(yy.rule_id);} ;

FOR__ty_sim_n__USE_  :
        FOR_TOKEN sim_n USE_TOKEN  {assign_association(yy.rule_id);} ;

.algt_cl.  :
        ..prag..
  |   ..prag.. algt_cl ..prag..   {assign_association(yy.rule_id);} ;

..cmpon_cl..  :
```

```
  |  ..cmpon_cl.. cmpon_cl ..prag..  {assign_association(yy.rule_id);} ;

ty_mk_rec_aggr  :  qualified_expr     {assign_association(yy.rule_id);} ;

%%

package parser is

   procedure yyparse;

   echo : boolean := false;
   number_of_errors : natural := 0;

end parser;
with ada_tokens, ada_goto, ada_shift_reduce, ada_lex, text_io;
use  ada_tokens, ada_goto, ada_shift_reduce, ada_lex, text_io;
package body parser is

   procedure yyerror(s: in string := "syntax error") is
   begin
         number_of_errors := number_of_errors + 1;
         put("<<< *** ");
         put_line(s);
   end yyerror;

##%procedure_parse

end parser;
```

-- **ADA_LEX.L**

```
--
--/*------------------------------------------------------------------------
--/* Lexical input for LEX for LALR(1) Grammar for ANSI Ada
--/*
--/*            Herman Fischer
--/*          Litton Data Systems
--/*            March 26, 1984
--/*
--/* Accompanies Public Domain YACC format Ada grammar
--/*
--/*
--/*
--/*
--/*
--/*
--/*
--/*------------------------------------------------------------------------

%START IDENT Z

A          [aA]
B          [bB]
C          [cC]
D          [dD]
E          [eE]
F          [fF]
G          [gG]
H          [hH]
I          [iI]
J          [jJ]
K          [kK]
L          [lL]
M          [mM]
N          [nN]
O          [oO]
P          [pP]
Q          [qQ]
R          [rR]
S          [sS]
T          [tT]
U          [uU]
V          [vV]
W          [wW]
X          [xX]
Y          [yY]
Z          [zZ]

%%
{A}{B}{O}{R}{T}          {ECHO; text_io.new_line; ENTER(Z); return(ABORT_TOKEN);}
{A}{B}{S}          {ECHO; text_io.new_line; ENTER(Z); return(ABS_TOKEN);}
{A}{C}{C}{E}{P}{T}          {ECHO; text_io.new_line; ENTER(Z); return(ACCEPT_TOKEN);}
{A}{C}{C}{E}{S}{S}          {ECHO; text_io.new_line; ENTER(Z); return(ACCESS_TOKEN);}
```

```
{A}{L}{L}              {ECHO; text_io.new_line; ENTER(Z); return(ALL_TOKEN);}
{A}{N}{D}              {ECHO; text_io.new_line; ENTER(Z); return(AND_TOKEN);}
{A}{R}{R}{A}{Y}              {ECHO; text_io.new_line; ENTER(Z); return(ARRAY_TOKEN);}
{A}{T}                 {ECHO; text_io.new_line; ENTER(Z); return(AT_TOKEN);}
{B}{E}{G}{I}{N}              {ECHO; text_io.new_line; ENTER(Z); return(BEGIN_TOKEN);}
{B}{O}{D}{Y}                 {ECHO; text_io.new_line; ENTER(Z); return(BODY_TOKEN);}
{C}{A}{S}{E}                 {ECHO; text_io.new_line; ENTER(Z); return(CASE_TOKEN);}
{C}{O}{N}{S}{T}{A}{N}{T}     {ECHO; text_io.new_line; ENTER(Z);
return(CONSTANT_TOKEN);}
{D}{E}{C}{L}{A}{R}{E}        {ECHO; text_io.new_line; ENTER(Z);
return(DECLARE_TOKEN);}
{D}{E}{L}{A}{Y}              {ECHO; text_io.new_line; ENTER(Z); return(DELAY_TOKEN);}
{D}{E}{L}{T}{A}              {ECHO; text_io.new_line; ENTER(Z); return(DELTA_TOKEN);}
{D}{I}{G}{I}{T}{S}           {ECHO; text_io.new_line; ENTER(Z); return(DIGITS_TOKEN);}
{D}{O}                 {ECHO; text_io.new_line; ENTER(Z); return(DO_TOKEN);}
{E}{L}{S}{E}           {ECHO; text_io.new_line; ENTER(Z); return(ELSE_TOKEN);}
{E}{L}{S}{I}{F}              {ECHO; text_io.new_line; ENTER(Z); return(ELSIF_TOKEN);}
{E}{N}{D}              {ECHO; text_io.new_line; ENTER(Z); return(END_TOKEN);}
{E}{N}{T}{R}{Y}              {ECHO; text_io.new_line; ENTER(Z); return(ENTRY_TOKEN);}
{E}{X}{C}{E}{P}{T}{I}{O}{N}        {ECHO; text_io.new_line; ENTER(Z);
return(EXCEPTION_TOKEN);}
{E}{X}{I}{T}           {ECHO; text_io.new_line; ENTER(Z); return(EXIT_TOKEN);}
{F}{O}{R}              {ECHO; text_io.new_line; ENTER(Z); return(FOR_TOKEN);}
{F}{U}{N}{C}{T}{I}{O}{N}     {ECHO; text_io.new_line; ENTER(Z);
return(FUNCTION_TOKEN);}
{G}{E}{N}{E}{R}{I}{C}  {ECHO; text_io.new_line; ENTER(Z); return(GENERIC_TOKEN);}
{G}{O}{T}{O}           {ECHO; text_io.new_line; ENTER(Z); return(GOTO_TOKEN);}
{I}{F}                 {ECHO; text_io.new_line; ENTER(Z); return(IF_TOKEN);}
{I}{N}                 {ECHO; text_io.new_line; ENTER(Z); return(IN_TOKEN);}
{I}{S}                 {ECHO; text_io.new_line; ENTER(Z); return(IS_TOKEN);}
{L}{I}{M}{I}{T}{E}{D}  {ECHO; text_io.new_line; ENTER(Z); return(LIMITED_TOKEN);}
{L}{O}{O}{P}           {ECHO; text_io.new_line; ENTER(Z); return(LOOP_TOKEN);}
{M}{O}{D}              {ECHO; text_io.new_line; ENTER(Z); return(MOD_TOKEN);}
{N}{E}{W}              {ECHO; text_io.new_line; ENTER(Z); return(NEW_TOKEN);}
{N}{O}{T}              {ECHO; text_io.new_line; ENTER(Z); return(NOT_TOKEN);}
{N}{U}{L}{L}           {ECHO; text_io.new_line; ENTER(Z); return(NULL_TOKEN);}
{O}{F}                 {ECHO; text_io.new_line; ENTER(Z); return(OF_TOKEN);}
{O}{R}                 {ECHO; text_io.new_line; ENTER(Z); return(OR_TOKEN);}
{O}{T}{H}{E}{R}{S}           {ECHO; text_io.new_line; ENTER(Z); return(OTHERS_TOKEN);}
{O}{U}{T}              {ECHO; text_io.new_line; ENTER(Z); return(OUT_TOKEN);}
{P}{A}{C}{K}{A}{G}{E}        {ECHO; text_io.new_line; ENTER(Z);
return(PACKAGE_TOKEN);}
{P}{R}{A}{G}{M}{A}           {ECHO; text_io.new_line; ENTER(Z); return(PRAGMA_TOKEN);}
{P}{R}{I}{V}{A}{T}{E}        {ECHO; text_io.new_line; ENTER(Z); return(PRIVATE_TOKEN);}
{P}{R}{O}{C}{E}{D}{U}{R}{E}        {ECHO; text_io.new_line; ENTER(Z);
return(PROCEDURE_TOKEN);}
{R}{A}{I}{S}{E}              {ECHO; text_io.new_line; ENTER(Z); return(RAISE_TOKEN);}
{R}{A}{N}{G}{E}              {ECHO; text_io.new_line; ENTER(Z); return(RANGE_TOKEN);}
{R}{E}{C}{O}{R}{D}           {ECHO; text_io.new_line; ENTER(Z); return(RECORD_TOKEN);}
{R}{E}{M}              {ECHO; text_io.new_line; ENTER(Z); return(REM_TOKEN);}
{R}{E}{N}{A}{M}{E}{S}        {ECHO; text_io.new_line; ENTER(Z);
return(RENAMES_TOKEN);}
{R}{E}{T}{U}{R}{N}           {ECHO; text_io.new_line; ENTER(Z); return(RETURN_TOKEN);}
```

```
{R}{E}{V}{E}{R}{S}{E}  {ECHO; text_io.new_line; ENTER(Z); return(REVERSE_TOKEN);}
{S}{E}{L}{E}{C}{T}        {ECHO; text_io.new_line; ENTER(Z); return(SELECT_TOKEN);}
{S}{E}{P}{A}{R}{A}{T}{E}        {ECHO; text_io.new_line; ENTER(Z);
return(SEPARATE_TOKEN);}
{S}{U}{B}{T}{Y}{P}{E}  {ECHO; text_io.new_line; ENTER(Z); return(SUBTYPE_TOKEN);}
{T}{A}{S}{K}                {ECHO; text_io.new_line; ENTER(Z); return(TASK_TOKEN);}
{T}{E}{R}{M}{I}{N}{A}{T}{E}  {ECHO; text_io.new_line; ENTER(Z);
return(TERMINATE_TOKEN);}
{T}{H}{E}{N}                {ECHO; text_io.new_line; ENTER(Z); return(THEN_TOKEN);}
{T}{Y}{P}{E}                {ECHO; text_io.new_line; ENTER(Z); return(TYPE_TOKEN);}
{U}{S}{E}           {ECHO; text_io.new_line; ENTER(Z); return(USE_TOKEN);}
{W}{H}{E}{N}                {ECHO; text_io.new_line; ENTER(Z); return(WHEN_TOKEN);}
{W}{H}{I}{L}{E}             {ECHO; text_io.new_line; ENTER(Z); return(WHILE_TOKEN);}
{W}{I}{T}{H}                {ECHO; text_io.new_line; ENTER(Z); return(WITH_TOKEN);}
{X}{O}{R}           {ECHO; text_io.new_line; ENTER(Z); return(XOR_TOKEN);}
"=>"   {ECHO; text_io.new_line; ENTER(Z); return(ARROW);}
".."   {ECHO; text_io.new_line; ENTER(Z); return(DOUBLE_DOT);}
"**"   {ECHO; text_io.new_line; ENTER(Z); return(DOUBLE_STAR);}
":="   {ECHO; text_io.new_line; ENTER(Z); return(ASSIGNMENT);}
"/="   {ECHO; text_io.new_line; ENTER(Z); return(INEQUALITY);}
">="   {ECHO; text_io.new_line; ENTER(Z); return(GREATER_THAN_OR_EQUAL);}
"<="   {ECHO; text_io.new_line; ENTER(Z); return(LESS_THAN_OR_EQUAL);}
"<<"   {ECHO; text_io.new_line; ENTER(Z); return(LEFT_LABEL_BRACKET);}
">>"   {ECHO; text_io.new_line; ENTER(Z); return(RIGHT_LABEL_BRACKET);}
"<>"   {ECHO; text_io.new_line; ENTER(Z); return(BOX);}
"&"    {ECHO; text_io.new_line; ENTER(Z); return('&'); }
"("    {ECHO; text_io.new_line; ENTER(Z); return('('); }
")"    {ECHO; text_io.new_line; ENTER(IDENT); return(')'); }
"*"    {ECHO; text_io.new_line; ENTER(Z); return('*'); }
"+"    {ECHO; text_io.new_line; ENTER(Z); return('+'); }
","    {ECHO; text_io.new_line; ENTER(Z); return(','); }
"-"    {ECHO; text_io.new_line; ENTER(Z); return('-'); }
"."    {ECHO; text_io.new_line; ENTER(Z); return('.'); }
"/"    {ECHO; text_io.new_line; ENTER(Z); return('/'); }
":"    {ECHO; text_io.new_line; ENTER(Z); return(':'); }
";"    {ECHO; text_io.new_line; ENTER(Z); return(';'); }
"<"    {ECHO; text_io.new_line; ENTER(Z); return('<'); }
"="    {ECHO; text_io.new_line; ENTER(Z); return('='); }
">"    {ECHO; text_io.new_line; ENTER(Z); return('>'); }
"|"    {ECHO; text_io.new_line; ENTER(Z); return('|'); }
<IDENT>\'  {ECHO; text_io.new_line; ENTER(Z); return('''');}

[a-z_A-Z][a-z_A-Z0-9]*      {ECHO; text_io.new_line; ENTER(IDENT);return(IDENTIFIER);}
[0-9][0-9_]*([.][0-9_]+)?([Ee][-+]?[0-9_]+)?  {
                              ECHO; text_io.new_line; ENTER(Z);
                              return(INTEGER_LITERAL);}

[0-9][0-9_]*#[0-9a-fA-F_]+([.][0-9a-fA-F_]+)?#([Ee][-+]?[0-9_]+)? {
                              ECHO; text_io.new_line; ENTER(Z);
                              return(INTEGER_LITERAL);}

\"([^"]*(\"\")*)*\" {ECHO; text_io.new_line; ENTER(Z); return(STRING_LITERAL);}
```

```
<Z>\'([^']|\\')\'    {ECHO; text_io.new_line; ENTER(Z); return(CHARACTER_LITERAL);}

[ \t]   null;          -- ignore spaces and tabs (null was ECHO)

"--".*  null;          -- ignore comments to end-of-line (null was ECHO)

.            {ECHO;
             text_io.put_line("?? lexical error" & ada_lex_dfa.yytext & "??");
                num_errors := num_errors + 1;}
[\n]    { null; -- ECHO;  linenum;
}
%%

with ada_tokens;
use  ada_tokens;
use text_io;

package ada_lex is

 lines      : positive := 1;
 num_errors : natural := 0;

 procedure linenum;

 function yylex return token;

end ada_lex;

package body ada_lex is

 procedure linenum is
 begin
  text_io.put(integer'image(lines) & ":");
  lines := lines + 1;
 end linenum;

##

end ada_lex;
```

```
-- ADA_LEX_DFA.A
--
package ada_lex_dfa is
aflex_debug : boolean := false;
yytext_ptr : integer; -- points to start of yytext in buffer

-- yy_ch_buf has to be 2 characters longer than YY_BUF_SIZE because we need
-- to put in 2 end-of-buffer characters (this is explained where it is
-- done) at the end of yy_ch_buf
YY_READ_BUF_SIZE : constant integer :=  8192;
YY_BUF_SIZE : constant integer := YY_READ_BUF_SIZE * 2; -- size of input buffer
type unbounded_character_array is array(integer range <>) of character;
subtype ch_buf_type is unbounded_character_array(0..YY_BUF_SIZE + 1);
yy_ch_buf : ch_buf_type;
yy_cp, yy_bp : integer;

-- yy_hold_char holds the character lost when yytext is formed
yy_hold_char : character;
yy_c_buf_p : integer;   -- points to current character in buffer

function YYText return string;
function YYLength return integer;
procedure YY_DO_BEFORE_ACTION;
--These variables are needed between calls to YYLex.
yy_init : boolean := true; -- do we need to initialize YYLex?
yy_start : integer := 0; -- current start state number
subtype yy_state_type is integer;
yy_last_accepting_state : yy_state_type;
yy_last_accepting_cpos : integer;
end ada_lex_dfa;

with ada_lex_dfa; use ada_lex_dfa;
package body ada_lex_dfa is
function YYText return string is
   i : integer;
   str_loc : integer := 1;
   buffer : string(1..1024);
   EMPTY_STRING : constant string := "";
begin
   -- find end of buffer
   i := yytext_ptr;
   while ( yy_ch_buf(i) /= ASCII.NUL ) loop
   buffer(str_loc ) := yy_ch_buf(i);
      i := i + 1;
   str_loc := str_loc + 1;
   end loop;
--    return yy_ch_buf(yytext_ptr.. i - 1);

   if (str_loc < 2) then
      return EMPTY_STRING;
   else
```

```
    return buffer(1..str_loc-1);
  end if;

end;

-- returns the length of the matched text
function YYLength return integer is
begin
  return yy_cp - yy_bp;
end YYLength;

-- done after the current pattern has been matched and before the
-- corresponding action - sets up yytext

procedure YY_DO_BEFORE_ACTION is
begin
  yytext_ptr := yy_bp;
  yy_hold_char := yy_ch_buf(yy_cp);
  yy_ch_buf(yy_cp) := ASCII.NUL;
  yy_c_buf_p := yy_cp;
end YY_DO_BEFORE_ACTION;

end ada_lex_dfa;
```

```
-- ADA_LEX_IO.A
--
with ada_lex_dfa; use ada_lex_dfa;
with text_io; use text_io;
 ith text_io; use text_io; x_dfa;  e our newline, put it back on the end. ; max_size
package ada_lex_io is
NULL_IN_INPUT : exception;
AFLEX_INTERNAL_ERROR : exception;
UNEXPECTED_LAST_MATCH : exception;
PUSHBACK_OVERFLOW : exception;
AFLEX_SCANNER_JAMMED : exception;
type eob_action_type is ( EOB_ACT_RESTART_SCAN,
                          EOB_ACT_END_OF_FILE,
                          EOB_ACT_LAST_MATCH );
YY_END_OF_BUFFER_CHAR :  constant character:=  ASCII.NUL;
yy_n_chars : integer;       -- number of characters read into yy_ch_buf
 y_n_chars : integer;       -- number of characters read into yy_ch_buf . ; max_size
-- true when we've seen an EOF for the current input file
yy_eof_has_been_seen : boolean;
 y_eof_has_been_seen : boolean; or the current input file nto yy_ch_buf . ; max_size
procedure YY_INPUT(buf: out unbounded_character_array; result: out integer;
max_size: in integer);
function yy_get_next_buffer return eob_action_type;
procedure yyunput( c : character; yy_bp: in out integer );
procedure unput(c : character);
function input return character;
procedure output(c : character);
function yywrap return boolean;
procedure Open_Input(fname : in String);
procedure Close_Input;
procedure Create_Output(fname : in String := "");
procedure Close_Output;
end ada_lex_io;
 nd ada_lex_io; Output; fname : in String := ""); teger ); lt: out integer; max_size
package body ada_lex_io is
-- gets input and stuffs it into 'buf'.  number of characters read, or YY_NULL,
-- is returned in 'result'.
 - is returned in 'result'. into 'buf'.  number of characters read, or YY_NULL, size
procedure YY_INPUT(buf: out unbounded_character_array; result: out integer;
max_size: in integer) is
   c : character;
   i : integer := 1;
   loc : integer := buf'first;
begin
   while ( i <= max_size ) loop
   if (end_of_line) then -- Ada ate our newline, put it back on the end.
         buf(loc) := ASCII.LF;
         skip_line(1);
      else
         get(buf(loc));
   end if;
    nd if;  et(buf(loc)); II.LF; te our newline, put it back on the end. ; max_size
      loc := loc + 1;
```

```
   i := i + 1;
   end loop;
    nd loop; ; oc + 1; ); II.LF; te our newline, put it back on the end. ; max_size
   result := i - 1;
   exception
      when END_ERROR => result := i - 1;
   -- when we hit EOF we need to set yy_eof_has_been_seen
   yy_eof_has_been_seen := true;
end YY_INPUT;
 nd YY_INPUT; _been_seen := true; set yy_eof_has_been_seen ck on the end. ; max_size
-- yy_get_next_buffer - try to read in new buffer
--
-- returns a code representing an action
--    EOB_ACT_LAST_MATCH -
--    EOB_ACT_RESTART_SCAN - restart the scanner
--    EOB_ACT_END_OF_FILE - end of file
 -    EOB_ACT_END_OF_FILE - end of file  scanner een_seen ck on the end. ; max_size
function yy_get_next_buffer return eob_action_type is
   dest : integer := 0;
   source : integer := yytext_ptr - 1; -- copy prev. char, too
   number_to_move : integer;
   ret_val : eob_action_type;
   num_to_read : integer;
begin
   if ( yy_c_buf_p > yy_n_chars + 1 ) then
      raise NULL_IN_INPUT;
   end if;
   end if; e NULL_IN_INPUT; ars + 1 ) then opy prev. char, too  the end. ; max_size
   -- try to read more data
   -- try to read more data ars + 1 ) then opy prev. char, too  the end. ; max_size
   -- first move last chars to start of buffer
   number_to_move := yy_c_buf_p - yytext_ptr;
   number_to_move := yy_c_buf_p - yytext_ptr;  prev. char, too  the end. ; max_size
   for i in 0..number_to_move - 1 loop
      yy_ch_buf(dest) := yy_ch_buf(source);
   dest := dest + 1;
   source := source + 1;
   end loop;
       oop; source + 1;  yy_ch_buf(source);   prev. char, too  the end. ; max_size
   if ( yy_eof_has_been_seen ) then
   -- don't do the read, it's not guaranteed to return an EOF,
   -- just force an EOF
   -- just force an EOF  it's not guaranteed to return an EOF,  the end. ; max_size
   yy_n_chars := 0;
   else
   num_to_read := YY_BUF_SIZE - number_to_move - 1;
   num_to_read := YY_BUF_SIZE - number_to_move - 1; rn an EOF,  the end. ; max_size
   if ( num_to_read > YY_READ_BUF_SIZE ) then
      num_to_read := YY_READ_BUF_SIZE;
      end if;
      end if; ead := YY_READ_BUF_SIZE;  then  - 1; rn an EOF,  the end. ; max_size
   -- read in more data
```

```
   YY_INPUT( yy_ch_buf(number_to_move..yy_ch_buf'last), yy_n_chars, num_to_read
);
   end if;
   if ( yy_n_chars = 0 ) then
   if ( number_to_move = 1 ) then
      ret_val := EOB_ACT_END_OF_FILE;
   else
      ret_val := EOB_ACT_LAST_MATCH;
      end if;
      end if; := EOB_ACT_LAST_MATCH;  yy_ch_buf'last), yy_n_chars, num_to_read )ze
   yy_eof_has_been_seen := true;
   else
   ret_val := EOB_ACT_RESTART_SCAN;
   end if;
    nd if; := EOB_ACT_RESTART_SCAN; ;  yy_ch_buf'last), yy_n_chars, num_to_read )ze
   yy_n_chars := yy_n_chars + number_to_move;
   yy_ch_buf(yy_n_chars) := YY_END_OF_BUFFER_CHAR;
   yy_ch_buf(yy_n_chars + 1) := YY_END_OF_BUFFER_CHAR;
   yy_ch_buf(yy_n_chars + 1) := YY_END_OF_BUFFER_CHAR;  yy_n_chars, num_to_read )ze
   -- yytext begins at the second character in
   -- yy_ch_buf; the first character is the one which
   -- preceded it before reading in the latest buffer;
   -- it needs to be kept around in case it's a
   -- newline, so yy_get_previous_state() will have
   -- with '^' rules active
   -- with '^' rules active vious_state() will have r;  yy_n_chars, num_to_read )ze
   yytext_ptr := 1;
   yytext_ptr := 1;  active vious_state() will have r;  yy_n_chars, num_to_read )ze
   return ret_val;
end yy_get_next_buffer;
 nd yy_get_next_buffer; tive vious_state() will have r;  yy_n_chars, num_to_read )ze
procedure yyunput( c : character; yy_bp: in out integer ) is
   number_to_move : integer;
   dest : integer;
   source : integer;
   tmp_yy_cp : integer;
begin
   tmp_yy_cp := yy_c_buf_p;
   yy_ch_buf(tmp_yy_cp) := yy_hold_char; -- undo effects of setting up yytext
   yy_ch_buf(tmp_yy_cp) := yy_hold_char; -- undo effects of setting up yytext d )ze
   if ( tmp_yy_cp < 2 ) then
   -- need to shift things up to make room
   number_to_move := yy_n_chars + 2; -- +2 for EOB chars
   dest := YY_BUF_SIZE + 2;
   source := number_to_move;
   source := number_to_move; rs + 2; -- +2 for EOB chars of setting up yytext d )ze
   while ( source > 0 ) loop
      dest := dest - 1;
      source := source - 1;
         yy_ch_buf(dest) := yy_ch_buf(source);
   end loop;
   end loop; _ch_buf(dest) := yy_ch_buf(source); B chars of setting up yytext d )ze
   tmp_yy_cp := tmp_yy_cp + dest - source;
```

```
   yy_bp := yy_bp + dest - source;
   yy_n_chars := YY_BUF_SIZE;
   yy_n_chars := YY_BUF_SIZE; rce; source; rce); B chars of setting up yytext d )ze
   if ( tmp_yy_cp < 2 ) then
      raise PUSHBACK_OVERFLOW;
   end if;
   end if;
   end if; e PUSHBACK_OVERFLOW; e; source; rce); B chars of setting up yytext d )ze
   if ( tmp_yy_cp > yy_bp and then yy_ch_buf(tmp_yy_cp-1) = ASCII.LF ) then
   yy_ch_buf(tmp_yy_cp-2) := ASCII.LF;
   end if;
   end if; f(tmp_yy_cp-2) := ASCII.LF; h_buf(tmp_yy_cp-1) = ASCII.LF ) then t d )ze
   tmp_yy_cp := tmp_yy_cp - 1;
   yy_ch_buf(tmp_yy_cp) := c;
   yy_ch_buf(tmp_yy_cp) := c;  CII.LF; h_buf(tmp_yy_cp-1) = ASCII.LF ) then t d )ze
-- Note:  this code is the text of YY_DO_BEFORE_ACTION, only
--        here we get different yy_cp and yy_bp's
   yytext_ptr := yy_bp;
   yy_hold_char := yy_ch_buf(tmp_yy_cp);
   yy_ch_buf(tmp_yy_cp) := ASCII.NUL;
   yy_c_buf_p := tmp_yy_cp;
end yyunput;
 nd yyunput; p := tmp_yy_cp; SCII.NUL; );  yy_bp's TION, only SCII.LF ) then t d )ze
procedure unput(c : character) is
begin
    yyunput( c, yy_bp );
end unput;
 nd unput; t( c, yy_bp ); ter) is NUL; );  yy_bp's TION, only SCII.LF ) then t d )ze
function input return character is
   c : character;
   yy_cp : integer := yy_c_buf_p;
begin
   yy_ch_buf(yy_cp) := yy_hold_char;
   yy_ch_buf(yy_cp) := yy_hold_char;  );  yy_bp's TION, only SCII.LF ) then t d )ze
   if ( yy_ch_buf(yy_c_buf_p) = YY_END_OF_BUFFER_CHAR ) then
   -- need more input
   yytext_ptr := yy_c_buf_p;
   yy_c_buf_p := yy_c_buf_p + 1;
   yy_c_buf_p := yy_c_buf_p + 1; Y_END_OF_BUFFER_CHAR ) then SCII.LF ) then t d )ze
   case yy_get_next_buffer is
      -- this code, unfortunately, is somewhat redundant with
      -- that above
      -- that above unfortunately, is somewhat redundant with II.LF ) then t d )ze
      when EOB_ACT_END_OF_FILE =>
      if ( yywrap ) then
         yy_c_buf_p := yytext_ptr;
         return ASCII.NUL;
      end if;
      end if; rn ASCII.NUL; xt_ptr; s somewhat redundant with II.LF ) then t d )ze
      yy_ch_buf(0) := ASCII.LF;
      yy_n_chars := 1;
      yy_ch_buf(yy_n_chars) := YY_END_OF_BUFFER_CHAR;
      yy_ch_buf(yy_n_chars + 1) := YY_END_OF_BUFFER_CHAR;
```

```
        yy_eof_has_been_seen := false;
        yy_c_buf_p := 1;
        yytext_ptr := yy_c_buf_p;
        yy_hold_char := yy_ch_buf(yy_c_buf_p);
        yy_hold_char := yy_ch_buf(yy_c_buf_p); ) is EOF AR; ith II.LF ) then t d )ze
        return ( input );
        when EOB_ACT_RESTART_SCAN =>
        yy_c_buf_p := yytext_ptr;
        yy_c_buf_p := yytext_ptr; => c_buf_p); ) is EOF AR; ith II.LF ) then t d )ze
        when EOB_ACT_LAST_MATCH =>
        raise UNEXPECTED_LAST_MATCH;
        when others => null;
        end case;
    end if;
    end if; case; s => null; _MATCH; c_buf_p); ) is EOF AR; ith II.LF ) then t d )ze
    c := yy_ch_buf(yy_c_buf_p);
    yy_c_buf_p := yy_c_buf_p + 1;
    yy_hold_char := yy_ch_buf(yy_c_buf_p);
    yy_hold_char := yy_ch_buf(yy_c_buf_p); p); ) is EOF AR; ith II.LF ) then t d )ze
    return c;
end input;
 nd input; c; ar := yy_ch_buf(yy_c_buf_p); p); ) is EOF AR; ith II.LF ) then t d )ze
procedure output(c : character) is
begin
    text_io.put(c);
end output;
 nd output; put(c);  character) is buf_p); p); ) is EOF AR; ith II.LF ) then t d )ze
-- default yywrap function - always treat EOF as an EOF
function yywrap return boolean is
begin
    return true;
end yywrap;
 nd yywrap; rue; eturn boolean is s treat EOF as an EOF AR; ith II.LF ) then t d )ze
procedure Open_Input(fname : in String) is
    f : file_type;
begin
    yy_init := true;
    open(f, in_file, fname);
    set_input(f);
end Open_Input;
 nd Open_Input; ; e, fname); in String) is OF as an EOF AR; ith II.LF ) then t d )ze
procedure Create_Output(fname : in String := "") is
    f : file_type;
begin
    if (fname /= "") then
        create(f, out_file, fname);
        set_output(f);
    end if;
end Create_Output;
 nd Create_Output; f); ile, fname); tring := "") is EOF AR; ith II.LF ) then t d )ze
procedure Close_Input is
begin
    null;
```

```
end Close_Input;
 nd Close_Input;  tput is , fname); tring := "") is EOF AR; ith II.LF ) then t d )ze
procedure Close_Output is
begin
   null;
end Close_Output;
 nd Close_Output; tput is , fname); tring := "") is EOF AR; ith II.LF ) then t d )ze
end ada_lex_io;
 nd ada_lex_io; ; tput is , fname); tring := "") is EOF AR; ith II.LF ) then t d )ze
```

-- ADA_TOKENS.A

```
--
package Ada_Tokens is


  subtype yystype is integer;

   YYLVal, YYVal : YYSType;
   type Token is
      (End_Of_Input, Error, '&', ''',
       '(', ')', '*',
       '+', ',', '-',
       '.', '/', ':',
       ';', '<', '=',
       '>', '|', Arrow,
       Double_Dot, Double_Star, Assignment,
       Inequality, Greater_Than_Or_Equal, Less_Than_Or_Equal,
       Left_Label_Bracket, Right_Label_Bracket, Box,
       Abort_Token, Abs_Token, Accept_Token,
       Access_Token, All_Token, And_Token,
       Array_Token, At_Token, Begin_Token,
       Body_Token, Case_Token, Constant_Token,
       Declare_Token, Delay_Token, Delta_Token,
       Digits_Token, Do_Token, Else_Token,
       Elsif_Token, End_Token, Entry_Token,
       Exception_Token, Exit_Token, For_Token,
       Function_Token, Generic_Token, Goto_Token,
       If_Token, In_Token, Is_Token,
       Limited_Token, Loop_Token, Mod_Token,
       New_Token, Not_Token, Null_Token,
       Of_Token, Or_Token, Others_Token,
       Out_Token, Package_Token, Pragma_Token,
       Private_Token, Procedure_Token, Raise_Token,
       Range_Token, Record_Token, Rem_Token,
       Renames_Token, Return_Token, Reverse_Token,
       Select_Token, Separate_Token, Subtype_Token,
       Task_Token, Terminate_Token, Then_Token,
       Type_Token, Use_Token, When_Token,
       While_Token, With_Token, Xor_Token,
       Identifier, Integer_Literal, Real_Literal,
       Character_Literal, String_Literal, Error1,
       Error2, Error3, Error4,
       Error5, Error6, Error7,
       Error8, Error9, Error10,
       Error11, Error12, Error13,
       Error14, Error15 );

   Syntax_Error : exception;

end Ada_Tokens;
```

-- ADA_SHIFT_REDUCE.A

```
--
package Ada_Shift_Reduce is

   type Small_Integer is range -32_000 .. 32_000;

   type Shift_Reduce_Entry is record
      T   : Small_Integer;
      Act : Small_Integer;
   end record;
   pragma Pack(Shift_Reduce_Entry);

   subtype Row is Integer range -1 .. Integer'Last;

 --pragma suppress(index_check);

   type Shift_Reduce_Array is array (Row  range <>) of Shift_Reduce_Entry;

   Shift_Reduce_Matrix : constant Shift_Reduce_Array :=
      ( (-1,-1) -- Dummy Entry

-- state  0
,(-1,-301)
-- state  1
,( 52,-436),( 53,-436),( 68,-436)
,( 71,-436),( 80,-436),( 89,-436),(-1,-245)

-- state  2
,( 69, 7),(-1,-433)
-- state  3
,( 0,-1001),(-1,-1000)

-- state  4
,( 52, 22),( 53, 28),( 68, 20),( 71, 25)
,( 80, 24),(-1,-1000)
-- state  5
,( 89, 31),(-1,-256)

-- state  6
,(-1,-301)
-- state  7
,( 91, 34),(-1,-1000)
-- state  8
,(-1,-302)

-- state  9
,(-1,-1000)
-- state  10
,(-1,-248)
-- state  11
,(-1,-249)
-- state  12
,(-1,-250)
```

```
-- state  13
,(-1,-251)
-- state  14
,(-1,-252)
-- state  15
,(-1,-253)
-- state  16
,(-1,-254)

-- state  17
,( 13, 35),( 57, 36),(-1,-1000)
-- state  18
,( 13, 37)
,(-1,-1000)
-- state  19
,( 13, 38),(-1,-1000)
-- state  20
,( 37, 40)
,( 91, 39),(-1,-1000)
-- state  21
,( 61, 41),(-1,-1000)

-- state  22
,( 91, 42),( 95, 44),(-1,-1000)
-- state  23
,(-1,-255)

-- state  24
,( 4, 46),(-1,-1000)
-- state  25
,( 91, 47),(-1,-1000)

-- state  26
,( 52, 48),( 68, 49),( 71, 25),(-1,-1000)

-- state  27
,(-1,-435)
-- state  28
,(-1,-444)
-- state  29
,(-1,-246)
-- state  30
,(-1,-247)

-- state  31
,( 91, 53),(-1,-1000)
-- state  32
,(-1,-301)
-- state  33
,( 69, 7)
,(-1,-434)
-- state  34
```

```
,( 4, 57),(-1,-303)
-- state 35
,(-1,-200)

-- state 36
,( 61,-451),(-1,-301)
-- state 37
,(-1,-212)
-- state 38
,(-1,-267)

-- state 39
,( 57, 63),(-1,-1000)
-- state 40
,( 91, 53),(-1,-1000)

-- state 41
,( 91, 65),(-1,-1000)
-- state 42
,(-1,-203)
-- state 43
,(-1,-204)

-- state 44
,(-1,-205)
-- state 45
,( 4, 68),( 57, 70),(-1,-393)

-- state 46
,( 91, 65),(-1,-1000)
-- state 47
,( 4, 68),(-1,-393)

-- state 48
,( 91, 42),( 95, 44),(-1,-1000)
-- state 49
,( 91, 74)
,(-1,-1000)
-- state 50
,(-1,-268)
-- state 51
,(-1,-269)
-- state 52
,( 85, 76)
,( 89, 78),( 91, 79),(-1,-270)
-- state 53
,(-1,-107)

-- state 54
,(-1,-440)
-- state 55
,( 69, 7),(-1,-438)
-- state 56
```

```
,( 86, 82)
,(-1,-437)
-- state 57
,( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 91, 87),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 58
,( 13, 125),(-1,-1000)

-- state 59
,(-1,-399)
-- state 60
,( 1, 148),( 51, 160),( 52, 22)
,( 53, 28),( 68, 135),( 71, 25),( 81, 155)
,( 82, 136),( 85, 161),( 86, 82),( 91, 79)
,(-1,-84)
-- state 61
,( 69, 7),(-1,-331)
-- state 62
,( 36, 165)
,(-1,-1000)
-- state 63
,( 61, 167),(-1,-301)
-- state 64
,( 57, 168)
,(-1,-1000)
-- state 65
,(-1,-313)
-- state 66
,( 4, 170),( 10, 172)
,(-1,-452)
-- state 67
,(-1,-394)
-- state 68
,( 91, 79),(-1,-1000)

-- state 69
,( 77, 175),(-1,-1000)
-- state 70
,( 61, 176),(-1,-1000)

-- state 71
,( 5, 177),( 10, 172),(-1,-1000)
-- state 72
,(-1,-201)

-- state 73
,( 4, 68),(-1,-393)
-- state 74
,( 57, 178),(-1,-1000)

-- state 75
```

```
,( 12, 179),(-1,-1000)
-- state 76
,( 91, 180),(-1,-1000)

-- state 77
,(-1,-273)
-- state 78
,( 52, 48),( 71, 25),(-1,-1000)

-- state 79
,(-1,-309)
-- state 80
,(-1,-445)
-- state 81
,( 8, 184),( 13, 183)
,(-1,-1000)
-- state 82
,( 91, 65),(-1,-1000)
-- state 83
,(-1,-301)

-- state 84
,(-1,-305)
-- state 85
,( 5, 187),( 8, 188),(-1,-1000)

-- state 86
,(-1,-2)
-- state 87
,( 18, 189),(-1,-107)
-- state 88
,( 33, 190)
,(-1,-125)
-- state 89
,( 33, 191),(-1,-126)
-- state 90
,( 65, 192)
,(-1,-127)
-- state 91
,( 65, 193),(-1,-128)
-- state 92
,(-1,-129)

-- state 93
,( 33, 194),( 65, 195),( 90, 196),(-1,-341)

-- state 94
,( 90, 197),(-1,-342)
-- state 95
,( 14, 202),( 15, 200)
,( 16, 204),( 22, 201),( 23, 205),( 24, 203)
,( 56,-352),( 62, 199),(-1,-349)
-- state 96
```

```
,(-1,-131)

-- state 97
,( 2, 210),( 7, 208),( 9, 209),(-1,-132)

-- state 98
,(-1,-354)
-- state 99
,( 4, 123),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 100
,( 6, 213),( 11, 214),( 60, 215),( 75, 216)
,(-1,-133)
-- state 101
,(-1,-152)
-- state 102
,(-1,-153)
-- state 103
,(-1,-357)

-- state 104
,( 20, 218),(-1,-359)
-- state 105
,( 4, 123),( 61, 115)
,( 63, 108),( 91, 53),( 92, 114),( 93, 113)
,( 94, 118),( 95, 44),(-1,-1000)
-- state 106
,( 4, 123)
,( 61, 115),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 107
,(-1,-137)
-- state 108
,(-1,-138)
-- state 109
,(-1,-139)
-- state 110
,(-1,-140)

-- state 111
,( 3,-108),( 4,-108),( 10,-108),(-1,-141)

-- state 112
,(-1,-142)
-- state 113
,(-1,-4)
-- state 114
,(-1,-5)
-- state 115
,( 91, 65)
```

```
,(-1,-1000)
-- state 116
,(-1,-158)
-- state 117
,(-1,-101)
-- state 118
,(-1,-102)

-- state 119
,(-1,-103)
-- state 120
,(-1,-104)
-- state 121
,(-1,-105)
-- state 122
,(-1,-106)

-- state 123
,( 1, 229),( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 66, 228),( 91, 53),( 92, 114),( 93, 113)
,( 94, 118),( 95, 44),(-1,-1000)
-- state 124
,( 3, 233)
,( 4, 123),( 10, 232),(-1,-1000)
-- state 125
,(-1,-1)

-- state 126
,(-1,-96)
-- state 127
,(-1,-97)
-- state 128
,(-1,-86)
-- state 129
,(-1,-87)

-- state 130
,(-1,-88)
-- state 131
,(-1,-98)
-- state 132
,(-1,-99)
-- state 133
,(-1,-100)

-- state 134
,( 13, 35),( 57, 234),( 76, 235),(-1,-1000)

-- state 135
,( 37, 237),( 91, 236),(-1,-1000)
-- state 136
,( 37, 240)
```

```
,( 85, 238),(-1,-410)
-- state  137
,(-1,-6)
-- state  138
,(-1,-7)

-- state  139
,(-1,-8)
-- state  140
,(-1,-9)
-- state  141
,(-1,-10)
-- state  142
,(-1,-11)

-- state  143
,(-1,-12)
-- state  144
,(-1,-13)
-- state  145
,(-1,-14)
-- state  146
,(-1,-15)

-- state  147
,(-1,-16)
-- state  148
,( 13, 241),(-1,-1000)
-- state  149
,(-1,-289)

-- state  150
,(-1,-290)
-- state  151
,( 12, 242),(-1,-1000)
-- state  152
,(-1,-24)

-- state  153
,(-1,-25)
-- state  154
,(-1,-26)
-- state  155
,( 91, 243),(-1,-1000)

-- state  156
,( 13, 244),(-1,-1000)
-- state  157
,(-1,-291)
-- state  158
,(-1,-292)

-- state  159
```

```
,(-1,-293)
-- state 160
,( 91, 53),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 161
,( 91, 249),(-1,-1000)
-- state 162
,( 4, 123)
,( 74, 251),(-1,-1000)
-- state 163
,(-1,-301)
-- state 164
,(-1,-301)

-- state 165
,(-1,-301)
-- state 166
,( 1, 148),( 51, 160),( 52, 22)
,( 53, 28),( 68, 259),( 70, 257),( 71, 25)
,( 81, 155),( 82, 260),( 85, 161),( 86, 82)
,( 91, 79),(-1,-402)
-- state 167
,( 91, 65),(-1,-1000)

-- state 168
,(-1,-301)
-- state 169
,(-1,-453)
-- state 170
,( 4, 123),( 7, 101)
,( 9, 102),( 29, 105),( 61, 115),( 62, 106)
,( 63, 108),( 91, 53),( 92, 114),( 93, 113)
,( 94, 118),( 95, 44),(-1,-1000)
-- state 171
,( 13, 272)
,(-1,-1000)
-- state 172
,( 91, 273),(-1,-1000)
-- state 173
,( 12, 274)
,(-1,-1000)
-- state 174
,(-1,-395)
-- state 175
,( 91, 65),(-1,-1000)

-- state 176
,( 91, 65),(-1,-1000)
-- state 177
,( 52, 48),( 68, 279)
,( 71, 25),( 82, 280),(-1,-1000)
-- state 178
,(-1,-301)
```

```
-- state 179
,( 56, 283),(-1,-397)
-- state 180
,( 4, 285),( 57, 286)
,(-1,-1000)
-- state 181
,( 57, 288),(-1,-448)
-- state 182
,( 8, 290)
,(-1,-23)
-- state 183
,(-1,-257)
-- state 184
,( 91, 53),(-1,-1000)

-- state 185
,( 10, 172),(-1,-408)
-- state 186
,( 69, 7),(-1,-439)

-- state 187
,(-1,-304)
-- state 188
,( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 91, 87),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 189
,( 4, 123),( 7, 101)
,( 9, 102),( 29, 105),( 61, 115),( 62, 106)
,( 63, 108),( 91, 53),( 92, 114),( 93, 113)
,( 94, 118),( 95, 44),(-1,-1000)
-- state 190
,( 4, 123)
,( 7, 101),( 9, 102),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 191
,( 84, 296),(-1,-1000)
-- state 192
,( 4, 123),( 7, 101)
,( 9, 102),( 29, 105),( 61, 115),( 62, 106)
,( 63, 108),( 91, 53),( 92, 114),( 93, 113)
,( 94, 118),( 95, 44),(-1,-1000)
-- state 193
,( 45, 298)
,(-1,-1000)
-- state 194
,( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 84, 300),( 91, 53),( 92, 114),( 93, 113)
```

,(94, 118),(95, 44),(-1,-1000)
-- state 195
,(4, 123)
,(7, 101),(9, 102),(29, 105),(45, 302)
,(61, 115),(62, 106),(63, 108),(91, 53)
,(92, 114),(93, 113),(94, 118),(95, 44)
,(-1,-1000)
-- state 196
,(4, 123),(7, 101),(9, 102)
,(29, 105),(61, 115),(62, 106),(63, 108)
,(91, 53),(92, 114),(93, 113),(94, 118)
,(95, 44),(-1,-1000)
-- state 197
,(4, 123),(7, 101)
,(9, 102),(29, 105),(61, 115),(62, 106)
,(63, 108),(91, 53),(92, 114),(93, 113)
,(94, 118),(95, 44),(-1,-1000)
-- state 198
,(4, 123)
,(7, 101),(9, 102),(29, 105),(61, 115)
,(62, 106),(63, 108),(91, 53),(92, 114)
,(93, 113),(94, 118),(95, 44),(-1,-1000)

-- state 199
,(-1,-353)
-- state 200
,(-1,-143)
-- state 201
,(-1,-144)
-- state 202
,(-1,-145)

-- state 203
,(-1,-146)
-- state 204
,(-1,-147)
-- state 205
,(-1,-148)
-- state 206
,(-1,-130)

-- state 207
,(56, 306),(-1,-1000)
-- state 208
,(-1,-149)
-- state 209
,(-1,-150)

-- state 210
,(-1,-151)
-- state 211
,(4, 123),(29, 105),(61, 115)
,(62, 106),(63, 108),(91, 53),(92, 114)

```
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 212
,(-1,-355)
-- state 213
,(-1,-154)
-- state 214
,(-1,-155)
-- state 215
,(-1,-156)

-- state 216
,(-1,-157)
-- state 217
,( 4, 123),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 218
,( 4, 123),( 61, 115),( 63, 108),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 219
,(-1,-134)
-- state 220
,(-1,-135)
-- state 221
,(-1,-136)

-- state 222
,( 10, 172),(-1,-38)
-- state 223
,( 3, 311),( 4, 123)
,(-1,-159)
-- state 224
,(-1,-121)
-- state 225
,(-1,-329)
-- state 226
,( 14, 202)
,( 15, 200),( 16, 204),( 17,-76),( 18,-76)
,( 19, 313),( 22, 201),( 23, 205),( 24, 203)
,( 56,-352),( 62, 199),(-1,-349)
-- state 227
,( 3,-108)
,( 4,-108),( 10,-108),( 73, 314),(-1,-141)

-- state 228
,(-1,-79)
-- state 229
,(-1,-80)
-- state 230
,(-1,-335)
```

```
-- state 231
,(-1,-109)

-- state 232
,( 32, 320),( 91, 53),( 94, 318),( 95, 44)
,(-1,-1000)
-- state 233
,( 4, 123),( 42, 324),( 43, 323)
,( 73, 325),( 91, 53),(-1,-1000)
-- state 234
,( 61,-451)
,( 80, 328),(-1,-301)
-- state 235
,( 91, 53),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 236
,( 57, 63),( 76, 330)
,(-1,-1000)
-- state 237
,( 91, 53),(-1,-1000)
-- state 238
,(-1,-411)

-- state 239
,( 91, 332),(-1,-1000)
-- state 240
,( 91, 53),(-1,-1000)

-- state 241
,(-1,-17)
-- state 242
,( 34, 335),( 39, 337),( 49, 339)
,( 91, 65),(-1,-1000)
-- state 243
,( 57, 340),(-1,-1000)

-- state 244
,(-1,-222)
-- state 245
,( 3, 341),( 4, 123),( 10, 232)
,(-1,-1000)
-- state 246
,(-1,-108)
-- state 247
,( 86, 342),(-1,-101)

-- state 248
,( 86, 343),(-1,-106)
-- state 249
,( 4, 285),( 13, 346)
,( 57, 344),(-1,-1000)
-- state 250
,( 13, 347),(-1,-1000)
```

```
-- state 251
,(-1,-301)
-- state 252
,( 69, 7),(-1,-333)
-- state 253
,( 52, 22)
,( 53, 28),( 68, 358),( 71, 25),( 82, 136)
,( 86, 82),(-1,-85)
-- state 254
,( 69, 7),(-1,-332)

-- state 255
,( 1, 361),( 69, 7),(-1,-364)
-- state 256
,( 49, 363)
,(-1,-383)
-- state 257
,(-1,-301)
-- state 258
,( 13, 35),( 57, 366)
,( 76, 235),(-1,-1000)
-- state 259
,( 91, 236),(-1,-1000)

-- state 260
,( 85, 238),(-1,-410)
-- state 261
,( 47, 367),(-1,-1000)

-- state 262
,( 4, 170),( 10, 172),(-1,-452)
-- state 263
,( 36, 369)
,(-1,-404)
-- state 264
,(-1,-285)
-- state 265
,(-1,-456)
-- state 266
,( 18, 371)
,(-1,-1000)
-- state 267
,(-1,-288)
-- state 268
,( 18,-286),(-1,-101)

-- state 269
,( 18,-287),(-1,-103)
-- state 270
,(-1,-458)
-- state 271
,(-1,-454)
```

```
-- state 272
,(-1,-282)
-- state 273
,(-1,-314)
-- state 274
,( 56, 374),( 67, 375)
,(-1,-397)
-- state 275
,( 5, 377),( 13, 378),(-1,-1000)

-- state 276
,(-1,-202)
-- state 277
,( 4, 170),( 10, 172),(-1,-452)

-- state 278
,( 57, 380),(-1,-1000)
-- state 279
,( 37, 40),(-1,-1000)

-- state 280
,( 37, 381),(-1,-1000)
-- state 281
,(-1,-261)
-- state 282
,(-1,-446)

-- state 283
,( 67, 382),(-1,-398)
-- state 284
,( 91, 65),(-1,-1000)

-- state 285
,( 91, 79),(-1,-1000)
-- state 286
,( 4, 387),( 31, 395)
,( 34, 396),( 42, 390),( 43, 389),( 58, 386)
,( 73, 388),(-1,-406)
-- state 287
,( 57, 399),(-1,-1000)

-- state 288
,( 27, 401),( 91, 53),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 289
,( 13, 402),(-1,-1000)
-- state 290
,( 91, 403)
,(-1,-1000)
-- state 291
,(-1,-441)
-- state 292
```

```
,( 8, 405),( 13, 404)
,(-1,-1000)
-- state 293
,(-1,-306)
-- state 294
,(-1,-3)
-- state 295
,(-1,-338)

-- state 296
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 297
,(-1,-340)
-- state 298
,( 4, 123),( 7, 101)
,( 9, 102),( 29, 105),( 61, 115),( 62, 106)
,( 63, 108),( 91, 53),( 92, 114),( 93, 113)
,( 94, 118),( 95, 44),(-1,-1000)
-- state 299
,(-1,-337)

-- state 300
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 301
,(-1,-339)
-- state 302
,( 4, 123),( 7, 101)
,( 9, 102),( 29, 105),( 61, 115),( 62, 106)
,( 63, 108),( 91, 53),( 92, 114),( 93, 113)
,( 94, 118),( 95, 44),(-1,-1000)
-- state 303
,(-1,-343)

-- state 304
,(-1,-344)
-- state 305
,(-1,-350)
-- state 306
,( 4, 123),( 7, 101)
,( 9, 102),( 29, 105),( 61, 115),( 62, 106)
,( 63, 108),( 91, 53),( 92, 114),( 93, 113)
,( 94, 118),( 95, 44),(-1,-1000)
-- state 307
,(-1,-356)

-- state 308
,(-1,-358)
```

```
-- state 309
,(-1,-360)
-- state 310
,(-1,-160)
-- state 311
,( 4, 123)
,(-1,-1000)
-- state 312
,( 17, 415),( 18, 414),(-1,-1000)

-- state 313
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 314
,( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 91, 53),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 315
,( 17,-77),( 18,-77)
,(-1,-124)
-- state 316
,( 5, 418),( 8, 419),(-1,-1000)

-- state 317
,(-1,-111)
-- state 318
,(-1,-112)
-- state 319
,(-1,-113)
-- state 320
,(-1,-114)

-- state 321
,(-1,-110)
-- state 322
,(-1,-116)
-- state 323
,(-1,-117)
-- state 324
,(-1,-118)

-- state 325
,(-1,-119)
-- state 326
,(-1,-115)
-- state 327
,(-1,-361)
-- state 328
,( 13, 420)
,(-1,-1000)
```

```
-- state 329
,( 13, 421),(-1,-108)
-- state 330
,( 91, 65)
,(-1,-1000)
-- state 331
,( 57, 423),(-1,-1000)
-- state 332
,( 57, 424)
,(-1,-412)
-- state 333
,( 57, 426),(-1,-1000)
-- state 334
,( 4, 123)
,( 42, 435),( 43, 434),( 73, 314),( 76, 437)
,(-1,-311)
-- state 335
,( 4, 438),(-1,-1000)
-- state 336
,( 21, 440)
,(-1,-307)
-- state 337
,( 21, 445),( 34, 335),( 91, 65)
,(-1,-1000)
-- state 338
,( 21, 440),(-1,-307)
-- state 339
,( 13, 448)
,( 76, 447),(-1,-1000)
-- state 340
,( 91, 65),(-1,-1000)

-- state 341
,( 42, 324),( 43, 323),( 73, 325),( 91, 53)
,(-1,-1000)
-- state 342
,( 35, 450),(-1,-459)
-- state 343
,( 4, 123)
,( 7, 101),( 9, 102),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 344
,( 4, 459),( 31, 395),( 34, 396),( 42, 435)
,( 43, 434),( 58, 386),( 61, 464),( 73, 314)
,( 74, 463),(-1,-406)
-- state 345
,( 13, 467),( 57, 466)
,(-1,-1000)
-- state 346
,(-1,-82)
-- state 347
```

```
,(-1,-295)
-- state 348
,( 35, 468)
,( 69, 7),(-1,-460)
-- state 349
,(-1,-462)
-- state 350
,(-1,-89)

-- state 351
,(-1,-90)
-- state 352
,(-1,-91)
-- state 353
,(-1,-92)
-- state 354
,(-1,-93)

-- state 355
,(-1,-94)
-- state 356
,(-1,-95)
-- state 357
,( 13, 35),( 57, 234)
,(-1,-1000)
-- state 358
,( 37, 237),( 91, 39),(-1,-1000)

-- state 359
,(-1,-301)
-- state 360
,( 25, 488),( 28, 494),( 30, 501)
,( 38, 499),( 41, 493),( 50, 490),( 54, 492)
,( 55, 498),( 63, 489),( 72, 495),( 77, 491)
,( 79, 506),( 91, 53),( 94, 118),( 95, 44)
,(-1,-373)
-- state 361
,( 13, 510),(-1,-1000)
-- state 362
,(-1,-301)

-- state 363
,(-1,-301)
-- state 364
,( 47, 514),(-1,-1000)
-- state 365
,( 1, 148)
,( 51, 160),( 52, 22),( 53, 28),( 68, 259)
,( 71, 25),( 81, 155),( 82, 260),( 85, 161)
,( 86, 82),( 91, 79),(-1,-403)
-- state 366
,(-1,-451)
```

```
-- state 367
,( 91, 53),(-1,-375)
-- state 368
,( 13, 517),(-1,-1000)

-- state 369
,(-1,-301)
-- state 370
,( 47, 519),(-1,-1000)
-- state 371
,( 4, 123)
,( 7, 101),( 9, 102),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 372
,( 5, 521),( 8, 522),(-1,-1000)
-- state 373
,(-1,-208)

-- state 374
,( 67, 523),(-1,-398)
-- state 375
,(-1,-210)
-- state 376
,( 91, 65)
,(-1,-1000)
-- state 377
,(-1,-206)
-- state 378
,( 91, 79),(-1,-1000)

-- state 379
,( 13, 526),(-1,-1000)
-- state 380
,(-1,-301)
-- state 381
,( 91, 53)
,(-1,-1000)
-- state 382
,(-1,-447)
-- state 383
,( 21, 440),(-1,-307)

-- state 384
,( 12, 529),(-1,-1000)
-- state 385
,(-1,-325)
-- state 386
,(-1,-407)

-- state 387
,( 27, 531),(-1,-1000)
```

```
-- state 388
,( 27, 532),(-1,-1000)

-- state 389
,( 27, 533),(-1,-1000)
-- state 390
,( 27, 534),(-1,-1000)

-- state 391
,(-1,-279)
-- state 392
,(-1,-280)
-- state 393
,(-1,-58)
-- state 394
,(-1,-59)

-- state 395
,( 91, 65),(-1,-1000)
-- state 396
,( 4, 536),(-1,-1000)

-- state 397
,( 70, 537),(-1,-1000)
-- state 398
,( 13, 538),(-1,-1000)

-- state 399
,( 58, 386),(-1,-406)
-- state 400
,( 13,-449),(-1,-108)

-- state 401
,(-1,-450)
-- state 402
,(-1,-274)
-- state 403
,(-1,-310)
-- state 404
,(-1,-217)

-- state 405
,( 91, 65),(-1,-1000)
-- state 406
,(-1,-346)
-- state 407
,(-1,-348)

-- state 408
,(-1,-345)
-- state 409
,(-1,-347)
-- state 410
```

```
,( 3,-108),( 4,-108)
,( 5,-45),( 8,-45),( 10,-108),( 13,-45)
,( 17,-45),( 18,-45),( 21,-45),( 33,-45)
,( 57,-45),( 59,-45),( 65,-45),( 84,-45)
,( 90,-45),(-1,-141)
-- state 411
,( 19, 541),(-1,-1000)

-- state 412
,(-1,-351)
-- state 413
,(-1,-161)
-- state 414
,( 4, 123),( 7, 101)
,( 9, 102),( 29, 105),( 61, 115),( 62, 106)
,( 63, 108),( 91, 53),( 92, 114),( 93, 113)
,( 94, 118),( 95, 44),(-1,-1000)
-- state 415
,( 1, 229)
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 66, 228)
,( 91, 53),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 416
,( 17,-78),( 18,-78)
,(-1,-123)
-- state 417
,(-1,-44)
-- state 418
,(-1,-120)
-- state 419
,( 1, 229)
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 66, 228)
,( 91, 53),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 420
,(-1,-258)
-- state 421
,(-1,-221)

-- state 422
,( 10, 172),( 13, 547),(-1,-1000)
-- state 423
,( 80, 548)
,(-1,-301)
-- state 424
,(-1,-301)
-- state 425
,(-1,-223)
-- state 426
,( 80, 552)
,(-1,-301)
```

```
-- state 427
,(-1,-312)
-- state 428
,(-1,-39)
-- state 429
,(-1,-40)

-- state 430
,(-1,-41)
-- state 431
,(-1,-42)
-- state 432
,( 73, 314),(-1,-317)

-- state 433
,( 73, 314),(-1,-317)
-- state 434
,( 4, 123),( 7, 101)
,( 9, 102),( 29, 105),( 61, 115),( 62, 106)
,( 63, 108),( 91, 53),( 92, 114),( 93, 113)
,( 94, 118),( 95, 44),(-1,-1000)
-- state 435
,( 4, 123)
,( 7, 101),( 9, 102),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 436
,(-1,-37)
-- state 437
,( 91, 53),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 438
,( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 91, 53),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 439
,( 64, 562),(-1,-1000)

-- state 440
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 441
,( 13, 564),(-1,-1000)
-- state 442
,( 4, 123)
,( 42, 435),( 43, 434),( 73, 314),(-1,-311)

-- state 443
,( 21, 440),(-1,-307)
```

```
-- state 444
,( 21, 440),(-1,-307)

-- state 445
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 446
,( 13, 568),(-1,-1000)
-- state 447
,( 91, 65)
,(-1,-1000)
-- state 448
,(-1,-262)
-- state 449
,( 13, 570),(-1,-1000)

-- state 450
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 451
,( 13, 572),(-1,-1000)
-- state 452
,(-1,-29)

-- state 453
,(-1,-30)
-- state 454
,(-1,-31)
-- state 455
,(-1,-32)
-- state 456
,(-1,-33)

-- state 457
,(-1,-34)
-- state 458
,(-1,-35)
-- state 459
,( 91, 574),( 94, 575)
,(-1,-1000)
-- state 460
,(-1,-51)
-- state 461
,(-1,-52)
-- state 462
,(-1,-53)

-- state 463
,(-1,-301)
```

```
-- state 464
,( 91, 65),(-1,-1000)
-- state 465
,( 13, 580)
,(-1,-1000)
-- state 466
,( 4, 459),( 31, 395),( 34, 39[illegible]
,( 42, 435),( 43, 434),( 58, 386),( 61, 464)
,( 73, 314),( 74, 463),(-1,-406)
-- state 467
,(-1,-83)

-- state 468
,( 60, 582),(-1,-1000)
-- state 469
,(-1,-301)
-- state 470
,( 47, 585)
,( 91, 53),( 94, 118),( 95, 44),(-1,-1000)

-- state 471
,( 69, 7),(-1,-334)
-- state 472
,(-1,-167)
-- state 473
,(-1,-168)

-- state 474
,(-1,-169)
-- state 475
,(-1,-170)
-- state 476
,(-1,-171)
-- state 477
,(-1,-172)

-- state 478
,(-1,-173)
-- state 479
,(-1,-174)
-- state 480
,(-1,-175)
-- state 481
,( 13, 587)
,( 21, 588),(-1,-108)
-- state 482
,(-1,-177)
-- state 483
,(-1,-178)

-- state 484
,(-1,-179)
-- state 485
```

```
,(-1,-180)
-- state  486
,(-1,-181)
-- state  487
,(-1,-182)

-- state  488
,( 91, 53),(-1,-1000)
-- state  489
,( 13, 590),(-1,-1000)

-- state  490
,( 91, 65),(-1,-387)
-- state  491
,( 4, 123),( 7, 101)
,( 9, 102),( 29, 105),( 61, 115),( 62, 106)
,( 63, 108),( 91, 53),( 92, 114),( 93, 113)
,( 94, 118),( 95, 44),(-1,-391)
-- state  492
,( 91, 65)
,(-1,-1000)
-- state  493
,( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 91, 53),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state  494
,( 91, 53),( 94, 118)
,( 95, 44),(-1,-1000)
-- state  495
,( 91, 65),(-1,-387)

-- state  496
,( 13, 599),(-1,-1000)
-- state  497
,( 12, 600),(-1,-101)

-- state  498
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state  499
,( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 91, 53),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state  500
,( 40, 605),( 51, 607)
,( 59,-377),( 88, 606),(-1,-381)
-- state  501
,( 91, 53)
,(-1,-1000)
```

```
-- state 502
,(-1,-231)
-- state 503
,(-1,-232)
-- state 504
,(-1,-233)

-- state 505
,(-1,-464)
-- state 506
,( 87, 614),(-1,-301)
-- state 507
,(-1,-164)

-- state 508
,(-1,-165)
-- state 509
,(-1,-365)
-- state 510
,(-1,-166)
-- state 511
,( 69, 7)
,(-1,-362)
-- state 512
,( 1, 361),( 45,-162),( 46,-162)
,( 47,-162),( 49,-162),( 65,-162),( 87,-162)
,(-1,-364)
-- state 513
,( 69, 7),( 87, 627),(-1,-1000)

-- state 514
,( 91, 42),( 95, 44),(-1,-400)
-- state 515
,(-1,-376)

-- state 516
,(-1,-213)
-- state 517
,(-1,-281)
-- state 518
,( 49, 363),(-1,-383)

-- state 519
,( 91, 53),(-1,-375)
-- state 520
,(-1,-457)
-- state 521
,(-1,-284)

-- state 522
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
```

```
,(-1,-1000)
-- state 523
,(-1,-209)
-- state 524
,( 21, 440),(-1,-307)

-- state 525
,(-1,-396)
-- state 526
,(-1,-283)
-- state 527
,( 57, 634),(-1,-1000)

-- state 528
,( 13, 635),(-1,-1000)
-- state 529
,( 91, 65),(-1,-1000)

-- state 530
,( 5, 637),( 13, 638),(-1,-1000)
-- state 531
,( 5, 639)
,(-1,-1000)
-- state 532
,(-1,-276)
-- state 533
,(-1,-277)
-- state 534
,(-1,-278)

-- state 535
,(-1,-81)
-- state 536
,( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 91, 53),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 537
,( 13, 642),(-1,-1000)

-- state 538
,(-1,-272)
-- state 539
,( 70, 643),(-1,-1000)
-- state 540
,( 10, 172)
,(-1,-409)
-- state 541
,( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 91, 53),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 542
```

```
,(-1,-122)
-- state 543
,( 19, 645)
,(-1,-76)
-- state 544
,( 3,-108),( 4,-108),( 10,-108)
,( 73, 314),(-1,-141)
-- state 545
,(-1,-330)
-- state 546
,(-1,-336)

-- state 547
,(-1,-220)
-- state 548
,( 13, 647),(-1,-1000)
-- state 549
,( 69, 7)
,(-1,-414)
-- state 550
,( 48, 649),(-1,-416)
-- state 551
,( 36, 651)
,(-1,-1000)
-- state 552
,( 13, 652),(-1,-1000)
-- state 553
,(-1,-318)

-- state 554
,(-1,-54)
-- state 555
,(-1,-56)
-- state 556
,(-1,-55)
-- state 557
,(-1,-57)

-- state 558
,( 13, 653),(-1,-108)
-- state 559
,(-1,-64)
-- state 560
,( 3,-108)
,( 4,-108),( 5,-45),( 8,-45),( 10,-108)
,( 59,-45),( 73, 314),(-1,-141)
-- state 561
,(-1,-321)

-- state 562
,( 91, 65),(-1,-1000)
-- state 563
,(-1,-308)
```

```
-- state 564
,(-1,-18)

-- state 565
,( 13, 657),(-1,-1000)
-- state 566
,( 13, 658),(-1,-1000)

-- state 567
,( 13, 659),(-1,-1000)
-- state 568
,(-1,-20)
-- state 569
,( 10, 172)
,( 13, 660),(-1,-1000)
-- state 570
,(-1,-36)
-- state 571
,( 13, 661)
,(-1,-1000)
-- state 572
,(-1,-294)
-- state 573
,(-1,-48)
-- state 574
,(-1,-49)

-- state 575
,(-1,-50)
-- state 576
,(-1,-315)
-- state 577
,( 63, 664),( 69, 7)
,(-1,-323)
-- state 578
,( 47, 665),(-1,-1000)
-- state 579
,(-1,-43)

-- state 580
,(-1,-27)
-- state 581
,( 13, 666),(-1,-1000)
-- state 582
,( 4, 123)
,( 7, 101),( 9, 102),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 583
,( 69, 7),(-1,-461)
-- state 584
,( 35, 668),(-1,-108)
```

```
-- state 585
,( 74, 669),(-1,-1000)
-- state 586
,(-1,-301)
-- state 587
,(-1,-176)

-- state 588
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 589
,( 26, 672),(-1,-1000)
-- state 590
,(-1,-184)

-- state 591
,( 10, 172),(-1,-388)
-- state 592
,( 87, 673),(-1,-389)

-- state 593
,(-1,-392)
-- state 594
,( 13, 675),(-1,-1000)
-- state 595
,( 10, 172)
,( 13, 676),(-1,-1000)
-- state 596
,( 13, 677),(-1,-1000)

-- state 597
,( 8,-431),( 13,-431),(-1,-108)
-- state 598
,( 13, 679)
,(-1,-1000)
-- state 599
,(-1,-300)
-- state 600
,(-1,-374)
-- state 601
,(-1,-187)

-- state 602
,( 84, 680),(-1,-1000)
-- state 603
,( 57, 681),(-1,-1000)

-- state 604
,(-1,-378)
-- state 605
```

```
,(-1,-301)
-- state 606
,( 1, 684),( 4, 123)
,( 7, 101),( 9, 102),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 607
,( 1, 687),( 91, 685),(-1,-1000)
-- state 608
,( 59, 688)
,(-1,-1000)
-- state 609
,( 36, 689),(-1,-1000)
-- state 610
,( 4, 691)
,(-1,-393)
-- state 611
,(-1,-235)
-- state 612
,( 30, 501),( 41, 493)
,( 69, 7),( 83, 696),( 91, 53),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 613
,(-1,-424)
-- state 614
,( 4, 123)
,( 7, 101),( 9, 102),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 615
,(-1,-236)
-- state 616
,(-1,-237)
-- state 617
,(-1,-238)
-- state 618
,(-1,-239)

-- state 619
,(-1,-240)
-- state 620
,(-1,-241)
-- state 621
,(-1,-422)
-- state 622
,(-1,-301)

-- state 623
,(-1,-163)
-- state 624
,(-1,-301)
```

```
-- state 625
,(-1,-385)
-- state 626
,( 87, 627)
,(-1,-384)
-- state 627
,( 66, 705),( 91, 65),(-1,-1000)

-- state 628
,(-1,-401)
-- state 629
,( 13, 707),(-1,-1000)
-- state 630
,(-1,-405)

-- state 631
,( 13, 708),(-1,-1000)
-- state 632
,(-1,-455)
-- state 633
,(-1,-207)

-- state 634
,(-1,-301)
-- state 635
,(-1,-271)
-- state 636
,( 21, 440),(-1,-307)

-- state 637
,(-1,-72)
-- state 638
,( 91, 79),(-1,-1000)
-- state 639
,(-1,-275)

-- state 640
,( 3,-108),( 4,-108),( 5,-45),( 8,-45)
,( 10,-108),( 73, 711),(-1,-141)
-- state 641
,(-1,-319)

-- state 642
,(-1,-215)
-- state 643
,( 13, 713),(-1,-1000)
-- state 644
,(-1,-46)

-- state 645
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
```

```
,(-1,-1000)
-- state 646
,(-1,-77)
-- state 647
,(-1,-259)
-- state 648
,( 47, 715)
,( 51, 160),(-1,-1000)
-- state 649
,( 91, 717),(-1,-1000)

-- state 650
,(-1,-301)
-- state 651
,(-1,-301)
-- state 652
,(-1,-260)
-- state 653
,(-1,-218)

-- state 654
,(-1,-65)
-- state 655
,( 5, 720),( 8, 721),(-1,-1000)

-- state 656
,(-1,-61)
-- state 657
,(-1,-19)
-- state 658
,(-1,-21)
-- state 659
,(-1,-22)

-- state 660
,(-1,-219)
-- state 661
,(-1,-299)
-- state 662
,( 5, 722),( 8, 723)
,(-1,-1000)
-- state 663
,( 38, 725),( 91, 79),(-1,-1000)

-- state 664
,( 13, 728),(-1,-1000)
-- state 665
,( 74, 729),(-1,-1000)

-- state 666
,(-1,-28)
-- state 667
,( 13, 730),(-1,-1000)
```

```
-- state 668
,( 4, 123)
,( 7, 101),( 9, 102),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 669
,( 13, 732),(-1,-1000)
-- state 670
,( 69, 7),(-1,-463)

-- state 671
,( 13, 733),(-1,-1000)
-- state 672
,(-1,-183)
-- state 673
,( 4, 123)
,( 7, 101),( 9, 102),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 674
,( 13, 735),(-1,-1000)
-- state 675
,(-1,-198)
-- state 676
,(-1,-199)

-- state 677
,(-1,-230)
-- state 678
,( 8, 737),( 13, 736),(-1,-1000)

-- state 679
,(-1,-266)
-- state 680
,(-1,-301)
-- state 681
,(-1,-301)
-- state 682
,(-1,-382)

-- state 683
,(-1,-191)
-- state 684
,(-1,-192)
-- state 685
,( 56, 741),(-1,-1000)

-- state 686
,(-1,-193)
-- state 687
,(-1,-194)
```

```
-- state 688
,(-1,-301)
-- state 689
,(-1,-301)

-- state 690
,(-1,-418)
-- state 691
,( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 91, 745),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 692
,( 44, 747),(-1,-420)

-- state 693
,( 13, 749),(-1,-108)
-- state 694
,(-1,-301)
-- state 695
,(-1,-301)

-- state 696
,( 13, 752),(-1,-1000)
-- state 697
,( 18, 753),(-1,-1000)

-- state 698
,( 45, 754),( 65, 756),(-1,-368)
-- state 699
,( 1, 361)
,( 45,-429),( 47,-429),( 65,-429),( 69, 7)
,(-1,-364)
-- state 700
,(-1,-430)
-- state 701
,( 45, 757),( 65, 758)
,(-1,-1000)
-- state 702
,( 69, 7),(-1,-363)
-- state 703
,(-1,-386)

-- state 704
,( 10, 172),(-1,-264)
-- state 705
,(-1,-265)
-- state 706
,(-1,-442)

-- state 707
,(-1,-211)
-- state 708
```

```
,(-1,-214)
-- state 709
,(-1,-73)
-- state 710
,(-1,-326)

-- state 711
,( 4, 123),( 7, 101),( 9, 102),( 27, 760)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 91, 53),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 712
,( 5, 761),( 8, 762)
,(-1,-1000)
-- state 713
,(-1,-216)
-- state 714
,(-1,-78)
-- state 715
,( 91, 53)
,(-1,-375)
-- state 716
,(-1,-301)
-- state 717
,( 4, 765),(-1,-393)

-- state 718
,( 69, 7),(-1,-415)
-- state 719
,( 49, 363),(-1,-383)

-- state 720
,(-1,-63)
-- state 721
,( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 91, 53),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 722
,(-1,-47)
-- state 723
,( 91, 574)
,( 94, 575),(-1,-1000)
-- state 724
,( 12, 770),(-1,-1000)

-- state 725
,( 91, 53),(-1,-1000)
-- state 726
,(-1,-301)
-- state 727
,(-1,-301)
```

```
-- state 728
,(-1,-301)
-- state 729
,(-1,-66)
-- state 730
,(-1,-297)
-- state 731
,( 73, 775)
,(-1,-1000)
-- state 732
,(-1,-296)
-- state 733
,(-1,-185)
-- state 734
,(-1,-390)

-- state 735
,(-1,-197)
-- state 736
,(-1,-244)
-- state 737
,( 91, 53),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 738
,(-1,-366)
-- state 739
,( 69, 7)
,( 87, 778),(-1,-1000)
-- state 740
,( 47, 780),(-1,-1000)

-- state 741
,( 78, 781),(-1,-379)
-- state 742
,( 47, 783),(-1,-1000)

-- state 743
,( 49, 363),(-1,-383)
-- state 744
,(-1,-229)
-- state 745
,( 8,-309)
,( 12,-309),(-1,-107)
-- state 746
,( 5, 785),(-1,-1000)

-- state 747
,(-1,-301)
-- state 748
,( 13, 787),(-1,-1000)
-- state 749
,(-1,-227)
```

```
-- state 750
,(-1,-426)
-- state 751
,(-1,-427)
-- state 752
,(-1,-301)
-- state 753
,(-1,-301)

-- state 754
,(-1,-301)
-- state 755
,( 47, 792),(-1,-1000)
-- state 756
,( 87, 614)
,(-1,-301)
-- state 757
,(-1,-301)
-- state 758
,(-1,-301)
-- state 759
,( 17, 798)
,( 18, 797),(-1,-1000)
-- state 760
,(-1,-62)
-- state 761
,( 64, 799)
,(-1,-1000)
-- state 762
,( 91, 53),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 763
,(-1,-413)
-- state 764
,( 69, 7),(-1,-417)

-- state 765
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 91, 745)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 766
,( 13, 803),(-1,-1000)
-- state 767
,( 47, 804)
,(-1,-1000)
-- state 768
,(-1,-322)
-- state 769
,(-1,-316)
-- state 770
,( 91, 65)
,(-1,-1000)
```

```
-- state 771
,( 57, 807),(-1,-1000)
-- state 772
,( 47,-67)
,( 69, 7),( 87,-67),(-1,-324)
-- state 773
,( 69, 7)
,(-1,-68)
-- state 774
,( 69, 7),(-1,-69)
-- state 775
,( 4, 123)
,( 7, 101),( 9, 102),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 91, 53),( 92, 114)
,( 93, 113),( 94, 118),( 95, 44),(-1,-1000)

-- state 776
,( 8,-432),( 13,-432),(-1,-108)
-- state 777
,( 45, 754)
,( 46, 810),(-1,-368)
-- state 778
,( 1, 229),( 4, 123)
,( 7, 101),( 9, 102),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 66, 228),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 779
,(-1,-371)
-- state 780
,( 38, 813),(-1,-1000)

-- state 781
,(-1,-380)
-- state 782
,( 4, 123),( 7, 101),( 9, 102)
,( 29, 105),( 61, 115),( 62, 106),( 63, 108)
,( 91, 53),( 92, 114),( 93, 113),( 94, 118)
,( 95, 44),(-1,-1000)
-- state 783
,( 59, 815),(-1,-1000)

-- state 784
,( 47, 816),(-1,-1000)
-- state 785
,( 4, 68),(-1,-393)

-- state 786
,( 47, 818),(-1,-1000)
-- state 787
,(-1,-228)
-- state 788
,( 69, 7)
```

```
,(-1,-428)
-- state 789
,( 30, 501),( 41, 493),( 69, 7)
,( 83, 696),(-1,-1000)
-- state 790
,(-1,-425)
-- state 791
,(-1,-369)

-- state 792
,( 79, 819),(-1,-1000)
-- state 793
,(-1,-423)
-- state 794
,( 47, 820)
,(-1,-1000)
-- state 795
,( 41, 493),( 69, 7),(-1,-1000)

-- state 796
,( 47, 821),(-1,-1000)
-- state 797
,(-1,-301)
-- state 798
,( 66, 705)
,( 91, 65),(-1,-1000)
-- state 799
,( 91, 65),(-1,-1000)

-- state 800
,( 73, 825),(-1,-108)
-- state 801
,(-1,-320)
-- state 802
,( 5, 826)
,(-1,-1000)
-- state 803
,(-1,-225)
-- state 804
,( 91, 53),(-1,-375)

-- state 805
,(-1,-71)
-- state 806
,( 21, 440),(-1,-307)
-- state 807
,(-1,-301)

-- state 808
,( 13, 830),(-1,-1000)
-- state 809
,( 47, 831),(-1,-1000)
```

```
-- state 810
,( 4, 123),( 7, 101),( 9, 102),( 29, 105)
,( 61, 115),( 62, 106),( 63, 108),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 811
,(-1,-329)
-- state 812
,( 87, 778),(-1,-370)

-- state 813
,( 13, 835),(-1,-1000)
-- state 814
,(-1,-195)
-- state 815
,( 91, 53)
,(-1,-375)
-- state 816
,( 91, 53),(-1,-375)
-- state 817
,(-1,-419)

-- state 818
,( 91, 53),(-1,-375)
-- state 819
,( 13, 839),(-1,-1000)

-- state 820
,( 79, 840),(-1,-1000)
-- state 821
,( 79, 841),(-1,-1000)

-- state 822
,(-1,-263)
-- state 823
,(-1,-443)
-- state 824
,(-1,-60)
-- state 825
,( 27, 760)
,(-1,-1000)
-- state 826
,( 4, 68),(-1,-393)
-- state 827
,( 13, 843)
,(-1,-1000)
-- state 828
,( 13, 844),(-1,-1000)
-- state 829
,( 69, 7)
,( 87, 845),(-1,-1000)
-- state 830
,(-1,-298)
```

```
-- state 831
,( 55, 847)
,(-1,-1000)
-- state 832
,( 84, 848),(-1,-1000)
-- state 833
,( 17, 415)
,( 18, 849),(-1,-1000)
-- state 834
,(-1,-372)
-- state 835
,(-1,-188)

-- state 836
,( 13, 850),(-1,-1000)
-- state 837
,( 13, 851),(-1,-1000)

-- state 838
,(-1,-421)
-- state 839
,(-1,-234)
-- state 840
,( 13, 852),(-1,-1000)

-- state 841
,( 13, 853),(-1,-1000)
-- state 842
,( 13, 854),(-1,-1000)

-- state 843
,(-1,-224)
-- state 844
,(-1,-70)
-- state 845
,( 1, 229),( 4, 123)
,( 7, 101),( 9, 102),( 29, 105),( 61, 115)
,( 62, 106),( 63, 108),( 66, 228),( 91, 53)
,( 92, 114),( 93, 113),( 94, 118),( 95, 44)
,(-1,-1000)
-- state 846
,(-1,-327)
-- state 847
,( 13, 857),(-1,-1000)

-- state 848
,(-1,-301)
-- state 849
,(-1,-301)
-- state 850
,(-1,-190)
-- state 851
,(-1,-196)
```

```
-- state  852
,(-1,-242)
-- state  853
,(-1,-243)
-- state  854
,(-1,-226)
-- state  855
,(-1,-329)

-- state  856
,( 47, 861),( 87, 845),(-1,-1000)
-- state  857
,(-1,-186)

-- state  858
,(-1,-367)
-- state  859
,(-1,-189)
-- state  860
,( 17, 415),( 18, 863)
,(-1,-1000)
-- state  861
,( 38, 864),(-1,-1000)
-- state  862
,(-1,-328)

-- state  863
,(-1,-301)
-- state  864
,( 13, 866),(-1,-1000)
-- state  865
,(-1,-75)

-- state  866
,(-1,-74)
);
--  The offset vector
SHIFT_REDUCE_OFFSET : array (0.. 866) of Integer :=
( 0,
 1, 8, 10, 12, 18, 20, 21, 23, 24, 25,
 26, 27, 28, 29, 30, 31, 32, 35, 37, 39,
 42, 44, 47, 48, 50, 52, 56, 57, 58, 59,
 60, 62, 63, 65, 67, 68, 70, 71, 72, 74,
 76, 78, 79, 80, 81, 84, 86, 88, 91, 93,
 94, 95, 99, 100, 101, 103, 105, 118, 120, 121,
 133, 135, 137, 139, 141, 142, 145, 146, 148, 150,
 152, 155, 156, 158, 160, 162, 164, 165, 168, 169,
 170, 173, 175, 176, 177, 180, 181, 183, 185, 187,
 189, 191, 192, 196, 198, 207, 208, 212, 213, 224,
 229, 230, 231, 232, 234, 243, 252, 253, 254, 255,
 256, 260, 261, 262, 263, 265, 266, 267, 268, 269,
 270, 271, 272, 287, 291, 292, 293, 294, 295, 296,
```

297, 298, 299, 300, 304, 307, 310, 311, 312, 313,
314, 315, 316, 317, 318, 319, 320, 321, 323, 324,
325, 327, 328, 329, 330, 332, 334, 335, 336, 337,
341, 343, 346, 347, 348, 349, 362, 364, 365, 366,
379, 381, 383, 385, 386, 388, 390, 395, 396, 398,
401, 403, 405, 406, 408, 410, 412, 413, 426, 439,
452, 454, 467, 469, 483, 497, 510, 523, 536, 537,
538, 539, 540, 541, 542, 543, 544, 546, 547, 548,
549, 560, 561, 562, 563, 564, 565, 576, 585, 586,
587, 588, 590, 593, 594, 595, 607, 612, 613, 614,
615, 616, 621, 627, 630, 634, 637, 639, 640, 642,
644, 645, 650, 652, 653, 657, 658, 660, 662, 666,
668, 669, 671, 678, 680, 683, 685, 686, 690, 692,
694, 696, 699, 701, 702, 703, 705, 706, 708, 710,
711, 712, 713, 714, 717, 720, 721, 724, 726, 728,
730, 731, 732, 734, 736, 738, 746, 748, 753, 755,
757, 758, 761, 762, 763, 764, 777, 778, 791, 792,
805, 806, 819, 820, 821, 822, 835, 836, 837, 838,
839, 841, 844, 857, 870, 873, 876, 877, 878, 879,
880, 881, 882, 883, 884, 885, 886, 887, 889, 891,
893, 895, 897, 899, 905, 907, 909, 913, 915, 918,
920, 925, 927, 940, 950, 953, 954, 955, 958, 959,
960, 961, 962, 963, 964, 965, 966, 969, 972, 973,
989, 991, 992, 993, 995, 1007, 1008, 1010, 1012, 1013,
1015, 1028, 1031, 1032, 1034, 1035, 1037, 1038, 1040, 1042,
1043, 1045, 1046, 1048, 1050, 1051, 1052, 1054, 1056, 1058,
1060, 1061, 1062, 1063, 1064, 1066, 1068, 1070, 1072, 1074,
1076, 1077, 1078, 1079, 1080, 1082, 1083, 1084, 1085, 1086,
1102, 1104, 1105, 1106, 1119, 1134, 1137, 1138, 1139, 1154,
1155, 1156, 1159, 1161, 1162, 1163, 1165, 1166, 1167, 1168,
1169, 1170, 1172, 1174, 1187, 1200, 1201, 1205, 1218, 1220,
1233, 1235, 1240, 1242, 1244, 1257, 1259, 1261, 1262, 1264,
1277, 1279, 1280, 1281, 1282, 1283, 1284, 1285, 1286, 1289,
1290, 1291, 1292, 1293, 1295, 1297, 1307, 1308, 1310, 1311,
1316, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326,
1327, 1330, 1331, 1332, 1333, 1334, 1335, 1336, 1338, 1340,
1342, 1355, 1357, 1370, 1374, 1376, 1378, 1380, 1393, 1406,
1411, 1413, 1414, 1415, 1416, 1417, 1419, 1420, 1421, 1422,
1423, 1425, 1433, 1436, 1439, 1440, 1441, 1442, 1444, 1446,
1447, 1448, 1461, 1462, 1464, 1465, 1466, 1468, 1470, 1472,
1475, 1477, 1478, 1479, 1480, 1481, 1494, 1496, 1497, 1499,
1501, 1514, 1515, 1517, 1522, 1523, 1524, 1525, 1527, 1529,
1531, 1533, 1535, 1536, 1537, 1538, 1539, 1540, 1542, 1543,
1551, 1552, 1554, 1555, 1556, 1558, 1560, 1562, 1563, 1566,
1567, 1569, 1570, 1571, 1572, 1573, 1574, 1577, 1579, 1580,
1581, 1583, 1596, 1598, 1600, 1602, 1603, 1604, 1617, 1619,
1620, 1622, 1624, 1625, 1627, 1630, 1632, 1635, 1637, 1638,
1639, 1640, 1642, 1644, 1645, 1646, 1660, 1663, 1665, 1667,
1669, 1670, 1678, 1679, 1692, 1693, 1694, 1695, 1696, 1697,
1698, 1699, 1700, 1701, 1702, 1703, 1705, 1708, 1709, 1711,
1712, 1714, 1715, 1716, 1717, 1718, 1720, 1721, 1723, 1724,
1731, 1732, 1733, 1735, 1736, 1749, 1750, 1751, 1754, 1756,
1757, 1758, 1759, 1760, 1761, 1764, 1765, 1766, 1767, 1768,

```
   1769, 1770, 1773, 1776, 1778, 1780, 1781, 1783, 1796, 1798,
   1800, 1802, 1803, 1816, 1818, 1819, 1820, 1821, 1824, 1825,
   1826, 1827, 1828, 1829, 1830, 1832, 1833, 1834, 1835, 1836,
   1837, 1850, 1852, 1854, 1855, 1856, 1858, 1860, 1863, 1869,
   1870, 1873, 1875, 1876, 1878, 1879, 1880, 1881, 1882, 1883,
   1884, 1898, 1901, 1902, 1903, 1905, 1906, 1908, 1910, 1912,
   1913, 1926, 1927, 1930, 1932, 1934, 1935, 1936, 1937, 1938,
   1939, 1941, 1942, 1943, 1944, 1945, 1946, 1950, 1951, 1954,
   1956, 1958, 1960, 1962, 1963, 1966, 1968, 1969, 1971, 1972,
   1973, 1974, 1975, 1976, 1977, 1979, 1981, 1982, 1983, 1986,
   1987, 1989, 1993, 1994, 1996, 2009, 2011, 2013, 2014, 2015,
   2017, 2019, 2023, 2025, 2027, 2040, 2043, 2046, 2061, 2062,
   2064, 2065, 2078, 2080, 2082, 2084, 2086, 2087, 2089, 2094,
   2095, 2096, 2098, 2099, 2101, 2104, 2106, 2107, 2110, 2112,
   2114, 2115, 2117, 2118, 2120, 2121, 2123, 2124, 2126, 2128,
   2141, 2142, 2144, 2146, 2147, 2149, 2151, 2152, 2154, 2156,
   2158, 2160, 2161, 2162, 2163, 2165, 2167, 2169, 2171, 2174,
   2175, 2177, 2179, 2182, 2183, 2184, 2186, 2188, 2189, 2190,
   2192, 2194, 2196, 2197, 2198, 2213, 2214, 2216, 2217, 2218,
   2219, 2220, 2221, 2222, 2223, 2224, 2227, 2228, 2229, 2230,
   2233, 2235, 2236, 2237, 2239, 2240);
end Ada_Shift_Reduce;
```

```
-- ADA_GOTO.A
--
package Ada_Goto is

   type Small_Integer is range -32_000 .. 32_000;

   type Goto_Entry is record
      Nonterm  : Small_Integer;
      Newstate : Small_Integer;
   end record;

 --pragma suppress(index_check);

   subtype Row is Integer range -1 .. Integer'Last;

   type Goto_Parse_Table is array (Row range <>) of Goto_Entry;

   Goto_Matrix : constant Goto_Parse_Table :=
     ((-1,-1)  -- Dummy Entry.
-- State  0
,(-193, 1),(-61, 2),(-2, 3)
-- State  1
,(-201, 5)
,(-195, 4),(-194, 6)
-- State  2
,(-3, 8)
-- State  3

-- State  4
,(-215, 21)
,(-208, 26),(-207, 19),(-200, 23),(-199, 16)
,(-198, 15),(-197, 30),(-196, 29),(-165, 18)
,(-155, 17),(-84, 27),(-83, 14),(-17, 13)
,(-15, 12),(-13, 11),(-12, 10)
-- State  5
,(-202, 32)

-- State  6
,(-61, 33)
-- State  7

-- State  8

-- State  9

-- State  10

-- State  11

-- State  12

-- State  13
```

-- State 14

-- State 15

-- State 16

-- State 17

-- State 18

-- State 19

-- State 20

-- State 21

-- State 22
,(-157, 45),(-86, 43)
-- State 23

-- State 24

-- State 25

-- State 26
,(-165, 51)
,(-155, 50)
-- State 27

-- State 28
,(-209, 52)
-- State 29

-- State 30

-- State 31
,(-68, 54)
-- State 32
,(-241, 56)
,(-61, 55)
-- State 33
,(-3, 8)
-- State 34
,(-4, 58)
-- State 35

-- State 36
,(-163, 62)
,(-74, 60),(-73, 59),(-61, 61)
-- State 37

-- State 38

-- State 39

-- State 40
,(-68, 64)

-- State 41
,(-39, 66)
-- State 42

-- State 43

-- State 44

-- State 45
,(-158, 67),(-156, 69)
-- State 46
,(-39, 71)

-- State 47
,(-158, 67),(-156, 72)
-- State 48
,(-157, 73),(-86, 43)

-- State 49

-- State 50

-- State 51

-- State 52
,(-210, 80),(-27, 77),(-20, 75)
-- State 53

-- State 54
,(-203, 81)

-- State 55
,(-3, 8)
-- State 56
,(-79, 83)
-- State 57
,(-235, 94),(-234, 85)
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-102, 96),(-100, 93),(-99, 92)
,(-98, 91),(-97, 90),(-96, 89),(-95, 88)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-47, 95),(-46, 111)
,(-44, 112),(-7, 107),(-6, 86),(-5, 84)

-- State 58

-- State 59

```
-- State  60
,(-228, 162),(-227, 159),(-226, 158),(-225, 157)
,(-224, 150),(-223, 149),(-215, 21),(-208, 26)
,(-207, 19),(-170, 156),(-165, 18),(-155, 134)
,(-85, 133),(-84, 132),(-83, 131),(-82, 127)
,(-81, 126),(-79, 130),(-78, 129),(-77, 164)
,(-75, 163),(-27, 154),(-26, 153),(-25, 152)
,(-20, 151),(-19, 147),(-18, 146),(-17, 145)
,(-16, 144),(-15, 143),(-14, 142),(-13, 141)
,(-12, 140),(-11, 139),(-10, 138),(-9, 137)
,(-8, 128)
-- State  61
,(-3, 8)
-- State  62

-- State  63
,(-74, 166),(-61, 61)

-- State  64

-- State  65

-- State  66
,(-216, 169),(-214, 171)
-- State  67

-- State  68
,(-159, 174),(-20, 173)

-- State  69

-- State  70

-- State  71

-- State  72

-- State  73
,(-158, 67),(-156, 69)
-- State  74

-- State  75

-- State  76

-- State  77

-- State  78
,(-155, 181)
-- State  79
,(-24, 182)
```

-- State 80

-- State 81

-- State 82
,(-39, 185)
-- State 83
,(-61, 186)
-- State 84

-- State 85

-- State 86

-- State 87

-- State 88

-- State 89

-- State 90

-- State 91

-- State 92

-- State 93

-- State 94

-- State 95
,(-236, 207),(-111, 198)
,(-101, 206)
-- State 96

-- State 97
,(-112, 211)
-- State 98

-- State 99
,(-115, 116),(-110, 110)
,(-109, 109),(-107, 104),(-106, 103),(-105, 100)
,(-104, 212),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-46, 111)
,(-44, 112),(-7, 107)
-- State 100
,(-114, 217)
-- State 101

-- State 102

-- State 103

-- State 104
,(-108, 219)

-- State 105
,(-115, 116),(-110, 110),(-109, 109),(-107, 220)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-46, 111),(-44, 112)
,(-7, 107)
-- State 106
,(-115, 116),(-110, 110),(-109, 109)
,(-107, 221),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-46, 111)
,(-44, 112),(-7, 107)
-- State 107

-- State 108

-- State 109

-- State 110

-- State 111

-- State 112

-- State 113

-- State 114

-- State 115
,(-39, 222),(-37, 223)

-- State 116

-- State 117

-- State 118

-- State 119

-- State 120

-- State 121

-- State 122

-- State 123
,(-235, 94),(-115, 116),(-113, 99),(-110, 110)
,(-109, 109),(-107, 104),(-106, 103),(-105, 100)
,(-104, 98),(-103, 97),(-102, 96),(-100, 93)
,(-99, 92),(-98, 91),(-97, 90),(-96, 89)
,(-95, 88),(-93, 230),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-71, 225)

```
,(-68, 117),(-47, 226),(-46, 227),(-44, 112)
,(-7, 107),(-6, 224)
-- State 124
,(-44, 231)
-- State 125

-- State 126

-- State 127

-- State 128

-- State 129

-- State 130

-- State 131

-- State 132

-- State 133

-- State 134

-- State 135

-- State 136
,(-171, 239)

-- State 137

-- State 138

-- State 139

-- State 140

-- State 141

-- State 142

-- State 143

-- State 144

-- State 145

-- State 146

-- State 147

-- State 148
```

-- State 149

-- State 150

-- State 151

-- State 152

-- State 153

-- State 154

-- State 155

-- State 156

-- State 157

-- State 158

-- State 159

-- State 160
,(-90, 245),(-89, 248),(-88, 121),(-87, 120)
,(-86, 119),(-68, 247),(-46, 246)
-- State 161

-- State 162
,(-44, 250)

-- State 163
,(-76, 253),(-61, 252)
-- State 164
,(-61, 254)
-- State 165
,(-116, 256)
,(-61, 255)
-- State 166
,(-228, 162),(-227, 159),(-226, 158)
,(-225, 157),(-224, 150),(-223, 149),(-215, 21)
,(-208, 26),(-207, 19),(-170, 156),(-166, 261)
,(-165, 18),(-155, 258),(-79, 130),(-78, 129)
,(-77, 164),(-27, 154),(-26, 153),(-25, 152)
,(-20, 151),(-19, 147),(-18, 146),(-17, 145)
,(-16, 144),(-15, 143),(-14, 142),(-13, 141)
,(-12, 140),(-11, 139),(-10, 138),(-9, 137)
,(-8, 128)
-- State 167
,(-39, 262)
-- State 168
,(-163, 263),(-74, 60)
,(-73, 59),(-61, 61)
-- State 169

-- State 170
,(-235, 94),(-222, 267)
,(-221, 265),(-220, 266),(-219, 264),(-217, 271)
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-102, 96),(-100, 93),(-99, 92)
,(-98, 91),(-97, 90),(-96, 89),(-95, 88)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 269),(-68, 268),(-47, 95),(-46, 111)
,(-44, 112),(-7, 107),(-6, 270)
-- State 171

-- State 172

-- State 173

-- State 174
,(-160, 275)

-- State 175
,(-39, 222),(-37, 276)
-- State 176
,(-39, 277)
-- State 177
,(-155, 278)
,(-85, 133),(-84, 132),(-83, 131),(-81, 281)

-- State 178
,(-74, 166),(-61, 61)
-- State 179
,(-211, 284),(-162, 282)

-- State 180
,(-29, 287)
-- State 181
,(-213, 289)
-- State 182

-- State 183

-- State 184
,(-68, 291)
-- State 185
,(-169, 292)

-- State 186
,(-3, 8)
-- State 187

-- State 188
,(-235, 94),(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)

,(-105, 100),(-104, 98),(-103, 97),(-102, 96)
,(-100, 93),(-99, 92),(-98, 91),(-97, 90)
,(-96, 89),(-95, 88),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-47, 95),(-46, 111),(-44, 112),(-7, 107)
,(-6, 86),(-5, 293)
-- State 189
,(-235, 94),(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-102, 96),(-100, 93),(-99, 92),(-98, 91)
,(-97, 90),(-96, 89),(-95, 88),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-47, 95),(-46, 111),(-44, 112)
,(-7, 107),(-6, 294)
-- State 190
,(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-102, 96)
,(-100, 295),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-47, 95)
,(-46, 111),(-44, 112),(-7, 107)
-- State 191

-- State 192
,(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-102, 96),(-100, 297),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-47, 95),(-46, 111),(-44, 112),(-7, 107)

-- State 193

-- State 194
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-102, 96),(-100, 299),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-47, 95),(-46, 111),(-44, 112)
,(-7, 107)
-- State 195
,(-115, 116),(-113, 99),(-110, 110)
,(-109, 109),(-107, 104),(-106, 103),(-105, 100)
,(-104, 98),(-103, 97),(-102, 96),(-100, 301)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-47, 95),(-46, 111)
,(-44, 112),(-7, 107)
-- State 196
,(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-102, 96)
,(-100, 303),(-90, 124),(-89, 122),(-88, 121)

,(-87, 120),(-86, 119),(-68, 117),(-47, 95)
,(-46, 111),(-44, 112),(-7, 107)
-- State 197
,(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-102, 96),(-100, 304),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-47, 95),(-46, 111),(-44, 112),(-7, 107)

-- State 198
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-47, 305)
,(-46, 111),(-44, 112),(-7, 107)
-- State 199

-- State 200

-- State 201

-- State 202

-- State 203

-- State 204

-- State 205

-- State 206

-- State 207

-- State 208

-- State 209

-- State 210

-- State 211
,(-115, 116)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 307),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-46, 111),(-44, 112),(-7, 107)
-- State 212

-- State 213

-- State 214

-- State 215

-- State 216

-- State 217
,(-115, 116)
,(-110, 110),(-109, 109),(-107, 104),(-106, 308)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-46, 111),(-44, 112)
,(-7, 107)
-- State 218
,(-115, 116),(-110, 110),(-109, 109)
,(-107, 309),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-46, 111)
,(-44, 112),(-7, 107)
-- State 219

-- State 220

-- State 221

-- State 222

-- State 223
,(-44, 310)
-- State 224

-- State 225
,(-72, 312)

-- State 226
,(-236, 207),(-111, 198),(-101, 206)
-- State 227
,(-41, 315)

-- State 228

-- State 229

-- State 230
,(-94, 316)
-- State 231

-- State 232
,(-91, 321),(-86, 319),(-68, 317)

-- State 233
,(-92, 326),(-68, 322),(-44, 327)
-- State 234
,(-163, 62)
,(-74, 60),(-73, 59),(-61, 61)
-- State 235
,(-90, 245)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)

,(-68, 117),(-46, 329)
-- State 236

-- State 237
,(-68, 331)
-- State 238

-- State 239

-- State 240
,(-68, 333)

-- State 241

-- State 242
,(-39, 222),(-37, 334),(-23, 338),(-21, 336)

-- State 243

-- State 244

-- State 245
,(-44, 231)
-- State 246

-- State 247

-- State 248

-- State 249
,(-29, 345)
-- State 250

-- State 251
,(-229, 349),(-61, 348)

-- State 252
,(-3, 8)
-- State 253
,(-215, 21),(-208, 26),(-207, 19)
,(-170, 156),(-165, 18),(-155, 357),(-85, 133)
,(-84, 132),(-83, 131),(-82, 127),(-81, 126)
,(-80, 359),(-79, 355),(-75, 350),(-17, 356)
,(-15, 354),(-14, 353),(-13, 352),(-12, 351)

-- State 254
,(-3, 8)
-- State 255
,(-120, 360),(-117, 362),(-3, 8)

-- State 256
,(-151, 364)
-- State 257

,(-74, 365),(-61, 61)
-- State 258

-- State 259

-- State 260
,(-171, 239)

-- State 261

-- State 262
,(-216, 169),(-214, 368)
-- State 263
,(-167, 370)
-- State 264

-- State 265

-- State 266

-- State 267

-- State 268

-- State 269

-- State 270

-- State 271
,(-218, 372)

-- State 272

-- State 273

-- State 274
,(-162, 373),(-161, 376)
-- State 275

-- State 276

-- State 277
,(-216, 169),(-214, 379)

-- State 278

-- State 279

-- State 280

-- State 281

-- State 282

-- State 283

-- State 284
,(-39, 222),(-37, 383)
-- State 285
,(-66, 385),(-20, 384)

-- State 286
,(-212, 398),(-168, 397),(-54, 393),(-35, 392)
,(-33, 391),(-23, 394)
-- State 287

-- State 288
,(-90, 245),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-46, 400)
-- State 289

-- State 290

-- State 291

-- State 292

-- State 293

-- State 294

-- State 295

-- State 296
,(-115, 116),(-113, 99),(-110, 110)
,(-109, 109),(-107, 104),(-106, 103),(-105, 100)
,(-104, 98),(-103, 97),(-102, 96),(-100, 406)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-47, 95),(-46, 111)
,(-44, 112),(-7, 107)
-- State 297

-- State 298
,(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-102, 96)
,(-100, 407),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-47, 95)
,(-46, 111),(-44, 112),(-7, 107)
-- State 299

-- State 300
,(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)

,(-102, 96),(-100, 408),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-47, 95),(-46, 111),(-44, 112),(-7, 107)

-- State 301

-- State 302
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-102, 96),(-100, 409),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-47, 95),(-46, 111),(-44, 112)
,(-7, 107)
-- State 303

-- State 304

-- State 305

-- State 306
,(-115, 116),(-113, 99),(-110, 110)
,(-109, 109),(-107, 104),(-106, 103),(-105, 100)
,(-104, 98),(-103, 97),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-47, 411),(-46, 410),(-45, 412),(-44, 112)
,(-7, 107)
-- State 307

-- State 308

-- State 309

-- State 310

-- State 311
,(-44, 413)
-- State 312

-- State 313
,(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-47, 416),(-46, 111),(-44, 112)
,(-7, 107)
-- State 314
,(-115, 116),(-113, 99),(-110, 110)
,(-109, 109),(-107, 104),(-106, 103),(-105, 100)
,(-104, 98),(-103, 97),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-47, 411),(-46, 410),(-45, 417),(-44, 112)
,(-7, 107)
-- State 315

-- State 316

-- State 317

-- State 318

-- State 319

-- State 320

-- State 321

-- State 322

-- State 323

-- State 324

-- State 325

-- State 326

-- State 327

-- State 328

-- State 329

-- State 330
,(-39, 422)
-- State 331

-- State 332
,(-172, 425)
-- State 333

-- State 334
,(-53, 433)
,(-51, 432),(-44, 431),(-43, 430),(-42, 429)
,(-41, 428),(-40, 427),(-38, 436)
-- State 335
,(-57, 439)

-- State 336
,(-22, 441)
-- State 337
,(-39, 222),(-37, 442),(-23, 444)
,(-21, 443)
-- State 338
,(-22, 446)
-- State 339

-- State 340
,(-39, 222),(-37, 442)
,(-21, 449)
-- State 341
,(-92, 326),(-68, 322)
-- State 342

-- State 343
,(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-47, 451),(-46, 111)
,(-44, 112),(-7, 107)
-- State 344
,(-168, 397),(-54, 393)
,(-53, 433),(-51, 432),(-43, 462),(-42, 461)
,(-41, 460),(-36, 458),(-35, 457),(-34, 456)
,(-33, 455),(-32, 454),(-31, 453),(-30, 452)
,(-28, 465),(-23, 394)
-- State 345

-- State 346

-- State 347

-- State 348
,(-231, 469),(-3, 8)

-- State 349
,(-230, 470)
-- State 350

-- State 351

-- State 352

-- State 353

-- State 354

-- State 355

-- State 356

-- State 357

-- State 358

-- State 359
,(-61, 471)
-- State 360
,(-233, 496),(-180, 504)

,(-179, 503),(-178, 502),(-144, 500),(-138, 509)
,(-137, 487),(-136, 486),(-135, 485),(-134, 484)
,(-133, 483),(-132, 482),(-131, 480),(-130, 479)
,(-129, 478),(-128, 477),(-127, 476),(-126, 475)
,(-125, 474),(-124, 473),(-123, 472),(-122, 508)
,(-121, 507),(-115, 116),(-110, 505),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 497),(-46, 481)
-- State 361

-- State 362
,(-118, 512),(-61, 511)

-- State 363
,(-61, 513)
-- State 364

-- State 365
,(-228, 162),(-227, 159),(-226, 158)
,(-225, 157),(-224, 150),(-223, 149),(-215, 21)
,(-208, 26),(-207, 19),(-170, 156),(-165, 18)
,(-155, 258),(-79, 130),(-78, 129),(-77, 164)
,(-27, 154),(-26, 153),(-25, 152),(-20, 151)
,(-19, 147),(-18, 146),(-17, 145),(-16, 144)
,(-15, 143),(-14, 142),(-13, 141),(-12, 140)
,(-11, 139),(-10, 138),(-9, 137),(-8, 128)

-- State 366

-- State 367
,(-146, 516),(-68, 515)
-- State 368

-- State 369
,(-116, 518),(-61, 255)

-- State 370

-- State 371
,(-235, 94),(-222, 267),(-221, 520),(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-102, 96),(-100, 93),(-99, 92),(-98, 91)
,(-97, 90),(-96, 89),(-95, 88),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-47, 95),(-46, 111),(-44, 112)
,(-7, 107),(-6, 270)
-- State 372

-- State 373

-- State 374

-- State 375

-- State 376
,(-39, 222),(-37, 524)

-- State 377

-- State 378
,(-159, 525),(-20, 173)
-- State 379

-- State 380
,(-163, 62),(-74, 60)
,(-73, 59),(-61, 61)
-- State 381
,(-68, 527)
-- State 382

-- State 383
,(-22, 528)

-- State 384

-- State 385
,(-67, 530)
-- State 386

-- State 387

-- State 388

-- State 389

-- State 390

-- State 391

-- State 392

-- State 393

-- State 394

-- State 395
,(-39, 222),(-37, 442),(-21, 535)

-- State 396
,(-57, 439)
-- State 397

-- State 398

-- State 399

```
,(-168, 539)
-- State 400

-- State 401

-- State 402

-- State 403

-- State 404

-- State 405
,(-39, 540)
-- State 406

-- State 407

-- State 408

-- State 409

-- State 410

-- State 411

-- State 412

-- State 413

-- State 414
,(-235, 94)
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-102, 96),(-100, 93),(-99, 92)
,(-98, 91),(-97, 90),(-96, 89),(-95, 88)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-47, 95),(-46, 111)
,(-44, 112),(-7, 107),(-6, 542)
-- State 415
,(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-71, 545),(-68, 117),(-47, 543)
,(-46, 544),(-44, 112),(-7, 107)
-- State 416

-- State 417

-- State 418

-- State 419
,(-235, 94)
```

,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-102, 96),(-100, 93),(-99, 92)
,(-98, 91),(-97, 90),(-96, 89),(-95, 88)
,(-93, 546),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-71, 225),(-68, 117)
,(-47, 226),(-46, 227),(-44, 112),(-7, 107)
,(-6, 224)
-- State 420

-- State 421

-- State 422

-- State 423
,(-163, 263),(-74, 60),(-73, 59)
,(-61, 61)
-- State 424
,(-239, 550),(-61, 549)
-- State 425

-- State 426
,(-163, 551)
,(-74, 60),(-73, 59),(-61, 61)
-- State 427

-- State 428

-- State 429

-- State 430

-- State 431

-- State 432
,(-52, 554)
,(-41, 553)
-- State 433
,(-52, 555),(-41, 553)
-- State 434
,(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-47, 556),(-46, 111)
,(-44, 112),(-7, 107)
-- State 435
,(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-47, 557),(-46, 111),(-44, 112)
,(-7, 107)

-- State 436

-- State 437
,(-90, 245),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-46, 558)

-- State 438
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-58, 561)
,(-47, 411),(-46, 560),(-45, 559),(-44, 112)
,(-7, 107)
-- State 439

-- State 440
,(-235, 94),(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-102, 96)
,(-100, 93),(-99, 92),(-98, 91),(-97, 90)
,(-96, 89),(-95, 88),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-47, 95),(-46, 111),(-44, 112),(-7, 107)
,(-6, 563)
-- State 441

-- State 442
,(-53, 433),(-51, 432),(-44, 431)
,(-43, 430),(-42, 429),(-41, 428),(-40, 427)
,(-38, 436)
-- State 443
,(-22, 565)
-- State 444
,(-22, 566)
-- State 445
,(-235, 94)
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-102, 96),(-100, 93),(-99, 92)
,(-98, 91),(-97, 90),(-96, 89),(-95, 88)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-47, 95),(-46, 111)
,(-44, 112),(-7, 107),(-6, 567)
-- State 446

-- State 447
,(-39, 569)

-- State 448

-- State 449

-- State 450

,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-47, 571)
,(-46, 111),(-44, 112),(-7, 107)
-- State 451

-- State 452

-- State 453

-- State 454

-- State 455

-- State 456

-- State 457

-- State 458

-- State 459
,(-50, 573)
,(-48, 576)
-- State 460

-- State 461

-- State 462

-- State 463
,(-61, 577),(-60, 578)
-- State 464
,(-39, 222)
,(-37, 442),(-21, 579)
-- State 465

-- State 466
,(-168, 539),(-54, 393)
,(-53, 433),(-51, 432),(-43, 462),(-42, 461)
,(-41, 460),(-36, 458),(-35, 457),(-34, 456)
,(-33, 455),(-32, 454),(-31, 453),(-30, 452)
,(-28, 581),(-23, 394)
-- State 467

-- State 468

-- State 469
,(-61, 583)
-- State 470
,(-232, 586)
,(-90, 245),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-46, 584)

-- State 471
,(-3, 8)

-- State 472

-- State 473

-- State 474

-- State 475

-- State 476

-- State 477

-- State 478

-- State 479

-- State 480

-- State 481

-- State 482

-- State 483

-- State 484

-- State 485

-- State 486

-- State 487

-- State 488
,(-68, 589)
-- State 489

-- State 490
,(-152, 592),(-39, 591)
-- State 491
,(-235, 94)
,(-154, 594),(-115, 116),(-113, 99),(-110, 110)
,(-109, 109),(-107, 104),(-106, 103),(-105, 100)
,(-104, 98),(-103, 97),(-102, 96),(-100, 93)
,(-99, 92),(-98, 91),(-97, 90),(-96, 89)
,(-95, 88),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-47, 95)
,(-46, 111),(-44, 112),(-7, 107),(-6, 593)

-- State 492
,(-39, 595)

-- State 493
,(-115, 116),(-113, 99),(-110, 110)
,(-109, 109),(-107, 104),(-106, 103),(-105, 100)
,(-104, 98),(-103, 97),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-47, 596),(-46, 111),(-44, 112),(-7, 107)

-- State 494
,(-90, 245),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-46, 597)
-- State 495
,(-152, 598)
,(-39, 591)
-- State 496

-- State 497

-- State 498
,(-235, 94),(-139, 602),(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-102, 96),(-100, 93),(-99, 92),(-98, 91)
,(-97, 90),(-96, 89),(-95, 88),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-47, 95),(-46, 111),(-44, 112)
,(-7, 107),(-6, 601)
-- State 499
,(-235, 94),(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-102, 96),(-100, 93),(-99, 92),(-98, 91)
,(-97, 90),(-96, 89),(-95, 88),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-47, 95),(-46, 111),(-44, 112)
,(-7, 107),(-6, 603)
-- State 500
,(-150, 609),(-147, 604)
,(-145, 608)
-- State 501
,(-68, 610)
-- State 502

-- State 503

-- State 504

-- State 505

-- State 506
,(-190, 620),(-189, 619)
,(-188, 618),(-187, 617),(-186, 616),(-185, 615)
,(-184, 613),(-183, 611),(-181, 621),(-174, 622)
,(-61, 612)

-- State 507

-- State 508

-- State 509

-- State 510

-- State 511
,(-3, 8)
-- State 512
,(-120, 360),(-119, 623)
,(-117, 624)
-- State 513
,(-238, 626),(-204, 625),(-3, 8)

-- State 514
,(-164, 629),(-157, 628),(-86, 43)
-- State 515

-- State 516

-- State 517

-- State 518
,(-151, 630)

-- State 519
,(-146, 631),(-68, 515)
-- State 520

-- State 521

-- State 522
,(-235, 94),(-222, 267)
,(-221, 265),(-220, 266),(-219, 264),(-217, 632)
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-102, 96),(-100, 93),(-99, 92)
,(-98, 91),(-97, 90),(-96, 89),(-95, 88)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 269),(-68, 268),(-47, 95),(-46, 111)
,(-44, 112),(-7, 107),(-6, 270)
-- State 523

-- State 524
,(-22, 633)

-- State 525

-- State 526

-- State 527

-- State 528

-- State 529
,(-39, 222),(-37, 636)
-- State 530

-- State 531

-- State 532

-- State 533

-- State 534

-- State 535

-- State 536
,(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-58, 561),(-55, 641),(-47, 411)
,(-46, 640),(-45, 559),(-44, 112),(-7, 107)

-- State 537

-- State 538

-- State 539

-- State 540

-- State 541
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-47, 644)
,(-46, 111),(-44, 112),(-7, 107)
-- State 542

-- State 543

-- State 544
,(-41, 646)

-- State 545

-- State 546

-- State 547

-- State 548

-- State 549
,(-3, 8)
-- State 550
,(-240, 648),(-173, 650)
-- State 551

-- State 552

-- State 553

-- State 554

-- State 555

-- State 556

-- State 557

-- State 558

-- State 559

-- State 560
,(-41, 654)

-- State 561
,(-59, 655)
-- State 562
,(-39, 222),(-37, 442),(-21, 656)

-- State 563

-- State 564

-- State 565

-- State 566

-- State 567

-- State 568

-- State 569

-- State 570

-- State 571

-- State 572

-- State 573

-- State 574

-- State 575

-- State 576
,(-49, 662)
-- State 577
,(-62, 663),(-3, 8)
-- State 578

-- State 579

-- State 580

-- State 581

-- State 582
,(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-47, 667),(-46, 111)
,(-44, 112),(-7, 107)
-- State 583
,(-3, 8)
-- State 584

-- State 585

-- State 586
,(-61, 670)

-- State 587

-- State 588
,(-235, 94),(-115, 116),(-113, 99),(-110, 110)
,(-109, 109),(-107, 104),(-106, 103),(-105, 100)
,(-104, 98),(-103, 97),(-102, 96),(-100, 93)
,(-99, 92),(-98, 91),(-97, 90),(-96, 89)
,(-95, 88),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-47, 95)
,(-46, 111),(-44, 112),(-7, 107),(-6, 671)

-- State 589

-- State 590

-- State 591

-- State 592
,(-153, 674)
-- State 593

-- State 594

-- State 595

-- State 596

-- State 597
,(-192, 678)
-- State 598

-- State 599

-- State 600

-- State 601

-- State 602

-- State 603

-- State 604

-- State 605
,(-74, 60),(-73, 682)
,(-61, 61)
-- State 606
,(-235, 94),(-139, 683),(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-102, 96),(-100, 93),(-99, 92),(-98, 91)
,(-97, 90),(-96, 89),(-95, 88),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-47, 95),(-46, 111),(-44, 112)
,(-7, 107),(-6, 601)
-- State 607
,(-148, 686)
-- State 608

-- State 609

-- State 610
,(-175, 692)
,(-158, 67),(-156, 690)
-- State 611

-- State 612
,(-136, 694),(-128, 695)
,(-90, 245),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-46, 693),(-3, 8)

-- State 613

-- State 614

,(-235, 94),(-139, 697),(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-102, 96)
,(-100, 93),(-99, 92),(-98, 91),(-97, 90)
,(-96, 89),(-95, 88),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-47, 95),(-46, 111),(-44, 112),(-7, 107)
,(-6, 601)
-- State 615

-- State 616

-- State 617

-- State 618

-- State 619

-- State 620

-- State 621
,(-182, 698)
-- State 622
,(-191, 701),(-116, 700)
,(-61, 699)
-- State 623

-- State 624
,(-61, 702)
-- State 625

-- State 626
,(-204, 703)
-- State 627
,(-205, 706)
,(-39, 704)
-- State 628

-- State 629

-- State 630

-- State 631

-- State 632

-- State 633

-- State 634
,(-163, 551),(-74, 60),(-73, 59)
,(-61, 61)
-- State 635

```
-- State 636
,(-22, 709)
-- State 637

-- State 638
,(-66, 710),(-20, 384)

-- State 639

-- State 640
,(-41, 654)
-- State 641
,(-56, 712)
-- State 642

-- State 643

-- State 644

-- State 645
,(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-47, 714),(-46, 111),(-44, 112)
,(-7, 107)
-- State 646

-- State 647

-- State 648
,(-228, 162),(-227, 159),(-226, 158)
,(-225, 157),(-224, 150),(-223, 149),(-78, 716)

-- State 649

-- State 650
,(-61, 718)
-- State 651
,(-116, 719),(-61, 255)
-- State 652

-- State 653

-- State 654

-- State 655

-- State 656

-- State 657

-- State 658
```

-- State 659

-- State 660

-- State 661

-- State 662

-- State 663
,(-64, 727)
,(-63, 726),(-20, 724)
-- State 664

-- State 665

-- State 666

-- State 667

-- State 668
,(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-47, 731),(-46, 111),(-44, 112)
,(-7, 107)
-- State 669

-- State 670
,(-3, 8)
-- State 671

-- State 672

-- State 673
,(-235, 94),(-139, 734)
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-102, 96),(-100, 93),(-99, 92)
,(-98, 91),(-97, 90),(-96, 89),(-95, 88)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-47, 95),(-46, 111)
,(-44, 112),(-7, 107),(-6, 601)
-- State 674

-- State 675

-- State 676

-- State 677

-- State 678

-- State 679

-- State 680
,(-116, 738)
,(-61, 255)
-- State 681
,(-142, 740),(-61, 739)
-- State 682

-- State 683

-- State 684

-- State 685

-- State 686

-- State 687

-- State 688
,(-116, 742)
,(-61, 255)
-- State 689
,(-116, 743),(-61, 255)
-- State 690

-- State 691
,(-235, 94)
,(-177, 746),(-159, 174),(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-102, 96)
,(-100, 93),(-99, 92),(-98, 91),(-97, 90)
,(-96, 89),(-95, 88),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-47, 95),(-46, 111),(-44, 112),(-20, 173)
,(-7, 107),(-6, 744)
-- State 692
,(-176, 748)
-- State 693

-- State 694
,(-191, 750)
,(-116, 700),(-61, 699)
-- State 695
,(-191, 751),(-116, 700)
,(-61, 699)
-- State 696

-- State 697

-- State 698
,(-141, 755)

-- State 699
,(-120, 360),(-117, 362)
,(-3, 8)
-- State 700

-- State 701

-- State 702
,(-3, 8)
-- State 703

-- State 704

-- State 705

-- State 706
,(-206, 759)
-- State 707

-- State 708

-- State 709

-- State 710

-- State 711
,(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-47, 411),(-46, 410)
,(-45, 417),(-44, 112),(-7, 107)
-- State 712

-- State 713

-- State 714

-- State 715
,(-146, 763)
,(-68, 515)
-- State 716
,(-61, 764)
-- State 717
,(-158, 67),(-156, 766)

-- State 718
,(-3, 8)
-- State 719
,(-151, 767)
-- State 720

-- State 721

,(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-68, 117),(-58, 768),(-47, 411),(-46, 560)
,(-45, 559),(-44, 112),(-7, 107)
-- State 722

-- State 723
,(-50, 573)
,(-48, 769)
-- State 724

-- State 725
,(-68, 771)
-- State 726
,(-61, 772)
-- State 727
,(-61, 773)

-- State 728
,(-61, 774)
-- State 729

-- State 730

-- State 731

-- State 732

-- State 733

-- State 734

-- State 735

-- State 736

-- State 737
,(-90, 245),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-46, 776)

-- State 738
,(-140, 777)
-- State 739
,(-143, 779),(-3, 8)
-- State 740

-- State 741
,(-149, 782)

-- State 742

```
-- State 743
,(-151, 784)
-- State 744

-- State 745
,(-24, 182)
-- State 746

-- State 747
,(-116, 786),(-61, 255)

-- State 748

-- State 749

-- State 750

-- State 751

-- State 752
,(-61, 788)
-- State 753
,(-190, 620),(-189, 619),(-188, 618)
,(-187, 617),(-186, 616),(-185, 615),(-184, 790)
,(-61, 789)
-- State 754
,(-116, 791),(-61, 255)
-- State 755

-- State 756
,(-190, 620)
,(-189, 619),(-188, 618),(-187, 617),(-186, 616)
,(-185, 615),(-184, 613),(-183, 611),(-181, 793)
,(-61, 789)
-- State 757
,(-116, 794),(-61, 255)
-- State 758
,(-189, 619)
,(-186, 796),(-61, 795)
-- State 759

-- State 760

-- State 761

-- State 762
,(-90, 245),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-55, 801),(-46, 800)
-- State 763

-- State 764
,(-3, 8)
```

-- State 765
,(-159, 174)
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-58, 802)
,(-47, 411),(-46, 560),(-45, 559),(-44, 112)
,(-20, 173),(-7, 107)
-- State 766

-- State 767

-- State 768

-- State 769

-- State 770
,(-65, 806),(-39, 222)
,(-37, 442),(-21, 805)
-- State 771

-- State 772
,(-3, 8)
-- State 773
,(-3, 8)

-- State 774
,(-3, 8)
-- State 775
,(-115, 116),(-113, 99),(-110, 110)
,(-109, 109),(-107, 104),(-106, 103),(-105, 100)
,(-104, 98),(-103, 97),(-90, 124),(-89, 122)
,(-88, 121),(-87, 120),(-86, 119),(-68, 117)
,(-47, 411),(-46, 410),(-45, 808),(-44, 112)
,(-7, 107)
-- State 776

-- State 777
,(-141, 809)
-- State 778
,(-115, 116),(-113, 99)
,(-110, 110),(-109, 109),(-107, 104),(-106, 103)
,(-105, 100),(-104, 98),(-103, 97),(-90, 124)
,(-89, 122),(-88, 121),(-87, 120),(-86, 119)
,(-71, 811),(-68, 117),(-47, 543),(-46, 544)
,(-44, 112),(-7, 107)
-- State 779
,(-237, 812)
-- State 780

-- State 781

-- State 782

,(-115, 116)
,(-113, 99),(-110, 110),(-109, 109),(-107, 104)
,(-106, 103),(-105, 100),(-104, 98),(-103, 97)
,(-90, 124),(-89, 122),(-88, 121),(-87, 120)
,(-86, 119),(-68, 117),(-58, 814),(-47, 411)
,(-46, 560),(-45, 559),(-44, 112),(-7, 107)

-- State 783

-- State 784

-- State 785
,(-158, 67),(-156, 817)
-- State 786

-- State 787

-- State 788
,(-3, 8)
-- State 789
,(-136, 694)
,(-128, 695),(-3, 8)
-- State 790

-- State 791

-- State 792

-- State 793

-- State 794

-- State 795
,(-128, 695),(-3, 8)

-- State 796

-- State 797
,(-116, 822),(-61, 255)
-- State 798
,(-205, 823),(-39, 704)

-- State 799
,(-39, 222),(-37, 442),(-21, 824)
-- State 800

-- State 801

-- State 802

-- State 803

-- State 804

,(-146, 827)
,(-68, 515)
-- State 805

-- State 806
,(-22, 828)
-- State 807
,(-61, 829)
-- State 808

-- State 809

-- State 810
,(-235, 94)
,(-139, 832),(-115, 116),(-113, 99),(-110, 110)
,(-109, 109),(-107, 104),(-106, 103),(-105, 100)
,(-104, 98),(-103, 97),(-102, 96),(-100, 93)
,(-99, 92),(-98, 91),(-97, 90),(-96, 89)
,(-95, 88),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-68, 117),(-47, 95)
,(-46, 111),(-44, 112),(-7, 107),(-6, 601)

-- State 811
,(-72, 833)
-- State 812
,(-143, 834)
-- State 813

-- State 814

-- State 815
,(-146, 836),(-68, 515)

-- State 816
,(-146, 837),(-68, 515)
-- State 817

-- State 818
,(-146, 838),(-68, 515)

-- State 819

-- State 820

-- State 821

-- State 822

-- State 823

-- State 824

-- State 825

-- State 826
,(-158, 67),(-156, 842)
-- State 827

-- State 828

-- State 829
,(-69, 846),(-3, 8)

-- State 830

-- State 831

-- State 832

-- State 833

-- State 834

-- State 835

-- State 836

-- State 837

-- State 838

-- State 839

-- State 840

-- State 841

-- State 842

-- State 843

-- State 844

-- State 845
,(-115, 116),(-113, 99),(-110, 110),(-109, 109)
,(-107, 104),(-106, 103),(-105, 100),(-104, 98)
,(-103, 97),(-90, 124),(-89, 122),(-88, 121)
,(-87, 120),(-86, 119),(-71, 855),(-68, 117)
,(-47, 543),(-46, 544),(-44, 112),(-7, 107)

-- State 846
,(-70, 856)
-- State 847

-- State 848
,(-116, 858),(-61, 255)

```
-- State 849
,(-116, 859)
,(-61, 255)
-- State 850

-- State 851

-- State 852

-- State 853

-- State 854

-- State 855
,(-72, 860)
-- State 856
,(-69, 862)
-- State 857

-- State 858

-- State 859

-- State 860

-- State 861

-- State 862

-- State 863
,(-61, 577)
,(-60, 865)
-- State 864

-- State 865

-- State 866

);
-- The offset vector
GOTO_OFFSET : array (0.. 866) of Integer :=
( 0,
 3, 6, 7, 7, 23, 24, 25, 25, 25, 25,
 25, 25, 25, 25, 25, 25, 25, 25, 25, 25,
 25, 25, 27, 27, 27, 27, 29, 29, 30, 30,
 30, 31, 33, 34, 35, 35, 39, 39, 39, 39,
 40, 41, 41, 41, 41, 43, 44, 46, 48, 48,
 48, 48, 51, 51, 52, 53, 54, 84, 84, 84,
 121, 122, 122, 124, 124, 124, 126, 126, 128, 128,
 128, 128, 128, 130, 130, 130, 130, 130, 131, 132,
 132, 132, 133, 134, 134, 134, 134, 134, 134, 134,
 134, 134, 134, 134, 134, 137, 137, 138, 138, 154,
 155, 155, 155, 155, 156, 169, 182, 182, 182, 182,
```

182, 182, 182, 182, 182, 184, 184, 184, 184, 184,
184, 184, 184, 214, 215, 215, 215, 215, 215, 215,
215, 215, 215, 215, 215, 215, 216, 216, 216, 216,
216, 216, 216, 216, 216, 216, 216, 216, 216, 216,
216, 216, 216, 216, 216, 216, 216, 216, 216, 216,
223, 223, 224, 226, 227, 229, 261, 262, 266, 266,
299, 299, 299, 299, 300, 302, 303, 308, 310, 312,
313, 314, 314, 314, 315, 316, 317, 317, 346, 374,
395, 395, 416, 416, 437, 458, 479, 500, 519, 519,
519, 519, 519, 519, 519, 519, 519, 519, 519, 519,
519, 535, 535, 535, 535, 535, 535, 549, 562, 562,
562, 562, 562, 563, 563, 564, 567, 568, 568, 568,
569, 569, 572, 575, 579, 586, 586, 587, 587, 587,
588, 588, 592, 592, 592, 593, 593, 593, 593, 594,
594, 596, 597, 616, 617, 620, 621, 623, 623, 623,
624, 624, 626, 627, 627, 627, 627, 627, 627, 627,
627, 628, 628, 628, 630, 630, 630, 632, 632, 632,
632, 632, 632, 632, 634, 636, 642, 642, 649, 649,
649, 649, 649, 649, 649, 649, 670, 670, 691, 691,
712, 712, 733, 733, 733, 733, 753, 753, 753, 753,
753, 754, 754, 773, 793, 793, 793, 793, 793, 793,
793, 793, 793, 793, 793, 793, 793, 793, 793, 793,
794, 794, 795, 795, 803, 804, 805, 809, 810, 810,
813, 815, 815, 834, 850, 850, 850, 850, 852, 853,
853, 853, 853, 853, 853, 853, 853, 853, 853, 854,
886, 886, 888, 889, 889, 920, 920, 922, 922, 924,
924, 954, 954, 954, 954, 954, 956, 956, 958, 958,
962, 963, 963, 964, 964, 965, 965, 965, 965, 965,
965, 965, 965, 965, 965, 968, 969, 969, 969, 970,
970, 970, 970, 970, 970, 971, 971, 971, 971, 971,
971, 971, 971, 971, 999, 1019, 1019, 1019, 1019, 1049,
1049, 1049, 1049, 1053, 1055, 1055, 1059, 1059, 1059, 1059,
1059, 1059, 1061, 1063, 1082, 1101, 1101, 1108, 1129, 1129,
1157, 1157, 1165, 1166, 1167, 1195, 1195, 1196, 1196, 1196,
1215, 1215, 1215, 1215, 1215, 1215, 1215, 1215, 1215, 1217,
1217, 1217, 1217, 1219, 1222, 1222, 1238, 1238, 1238, 1239,
1247, 1248, 1248, 1248, 1248, 1248, 1248, 1248, 1248, 1248,
1248, 1248, 1248, 1248, 1248, 1248, 1248, 1248, 1249, 1249,
1251, 1280, 1281, 1300, 1307, 1309, 1309, 1309, 1338, 1366,
1369, 1370, 1370, 1370, 1370, 1370, 1381, 1381, 1381, 1381,
1381, 1382, 1385, 1388, 1391, 1391, 1391, 1391, 1392, 1394,
1394, 1394, 1427, 1427, 1428, 1428, 1428, 1428, 1428, 1430,
1430, 1430, 1430, 1430, 1430, 1430, 1452, 1452, 1452, 1452,
1452, 1471, 1471, 1471, 1472, 1472, 1472, 1472, 1472, 1473,
1475, 1475, 1475, 1475, 1475, 1475, 1475, 1475, 1475, 1475,
1476, 1477, 1480, 1480, 1480, 1480, 1480, 1480, 1480, 1480,
1480, 1480, 1480, 1480, 1480, 1480, 1481, 1483, 1483, 1483,
1483, 1483, 1502, 1503, 1503, 1503, 1504, 1504, 1532, 1532,
1532, 1532, 1533, 1533, 1533, 1533, 1533, 1534, 1534, 1534,
1534, 1534, 1534, 1534, 1534, 1537, 1566, 1567, 1567, 1567,
1570, 1570, 1580, 1580, 1609, 1609, 1609, 1609, 1609, 1609,
1609, 1610, 1613, 1613, 1614, 1614, 1615, 1617, 1617, 1617,
1617, 1617, 1617, 1617, 1621, 1621, 1622, 1622, 1624, 1624,

```
1625, 1626, 1626, 1626, 1626, 1645, 1645, 1645, 1652, 1652,
1653, 1655, 1655, 1655, 1655, 1655, 1655, 1655, 1655, 1655,
1655, 1655, 1655, 1658, 1658, 1658, 1658, 1658, 1677, 1677,
1678, 1678, 1678, 1707, 1707, 1707, 1707, 1707, 1707, 1707,
1709, 1711, 1711, 1711, 1711, 1711, 1711, 1711, 1713, 1715,
1715, 1746, 1747, 1747, 1750, 1753, 1753, 1753, 1754, 1757,
1757, 1757, 1758, 1758, 1758, 1758, 1759, 1759, 1759, 1759,
1759, 1779, 1779, 1779, 1779, 1781, 1782, 1784, 1785, 1786,
1786, 1807, 1807, 1809, 1809, 1810, 1811, 1812, 1813, 1813,
1813, 1813, 1813, 1813, 1813, 1813, 1813, 1820, 1821, 1823,
1823, 1824, 1824, 1825, 1825, 1826, 1826, 1828, 1828, 1828,
1828, 1828, 1829, 1837, 1839, 1839, 1849, 1851, 1854, 1854,
1854, 1854, 1862, 1862, 1863, 1886, 1886, 1886, 1886, 1886,
1890, 1890, 1891, 1892, 1893, 1913, 1913, 1914, 1934, 1935,
1935, 1935, 1956, 1956, 1956, 1958, 1958, 1958, 1959, 1962,
1962, 1962, 1962, 1962, 1962, 1964, 1964, 1966, 1968, 1971,
1971, 1971, 1971, 1971, 1973, 1973, 1974, 1975, 1975, 1975,
2004, 2005, 2006, 2006, 2006, 2008, 2010, 2010, 2012, 2012,
2012, 2012, 2012, 2012, 2012, 2012, 2014, 2014, 2014, 2016,
2016, 2016, 2016, 2016, 2016, 2016, 2016, 2016, 2016, 2016,
2016, 2016, 2016, 2016, 2016, 2036, 2037, 2037, 2039, 2041,
2041, 2041, 2041, 2041, 2041, 2042, 2043, 2043, 2043, 2043,
2043, 2043, 2043, 2045, 2045, 2045);

subtype Rule        is Natural;
subtype Nonterminal is Integer;

  Rule_Length : array (Rule range  0 ..  464) of Natural := ( 2,
 4, 1, 3, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1,
 2, 5, 6, 5, 6, 6, 2, 1,
 1, 1, 5, 6, 1, 1, 1, 1,
 1, 1, 1, 5, 2, 1, 1, 1,
 1, 1, 2, 2, 1, 3, 4, 1,
 1, 1, 1, 1, 1, 2, 2, 2,
 2, 1, 1, 7, 4, 3, 4, 1,
 2, 4, 4, 4, 4, 5, 1, 4,
 4, 9, 5, 1, 2, 3, 1, 1,
 2, 3, 4, 1, 3, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 2, 3, 1, 1,
 1, 1, 3, 1, 1, 1, 1, 4,
 1, 4, 3, 2, 1, 1, 1, 1,
 1, 2, 1, 1, 1, 2, 2, 2,
 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 2, 3,
 4, 0, 4, 2, 2, 2, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 2,
 1, 1, 1, 1, 1, 1, 3, 2,
 4, 9, 1, 7, 5, 8, 2, 2,
 2, 2, 4, 8, 4, 3, 3, 2,
```

```
3, 5, 1, 1, 1, 4, 5, 1,
2, 1, 9, 2, 7, 9, 6, 7,
4, 6, 6, 5, 4, 2, 4, 11,
4, 7, 3, 5, 1, 3, 1, 1,
1, 7, 1, 1, 1, 1, 1, 1,
1, 8, 8, 4, 1, 2, 2, 1,
1, 1, 1, 1, 1, 1, 1, 1,
4, 4, 6, 6, 5, 4, 5, 1,
1, 3, 2, 2, 2, 2, 6, 5,
1, 4, 3, 2, 2, 2, 1, 1,
7, 5, 7, 4, 1, 1, 1, 1,
1, 1, 1, 1, 1, 5, 3, 7,
4, 6, 6, 2, 0, 2, 0, 3,
1, 3, 0, 2, 0, 3, 0, 1,
1, 3, 0, 3, 0, 1, 0, 3,
0, 3, 0, 3, 0, 3, 0, 2,
0, 3, 1, 3, 1, 3, 0, 3,
3, 3, 3, 3, 1, 1, 3, 3,
4, 4, 4, 4, 0, 2, 4, 0,
1, 1, 2, 3, 1, 3, 0, 2,
3, 1, 3, 0, 2, 0, 5, 0,
2, 3, 0, 2, 0, 2, 0, 1,
0, 1, 0, 1, 0, 2, 0, 3,
1, 2, 0, 1, 0, 2, 0, 1,
0, 1, 0, 3, 0, 1, 1, 0,
1, 0, 2, 0, 3, 0, 1, 0,
3, 0, 1, 0, 5, 1, 3, 0,
3, 1, 4, 0, 4, 0, 3, 1,
4, 3, 3, 4, 1, 1, 0, 3,
1, 3, 1, 0, 3, 1, 3, 0,
3, 0, 3, 0, 2, 1, 2, 0,
2, 2, 2, 0, 1, 0, 3, 1,
3, 1, 3, 1, 3, 0, 3, 1);
 Get_LHS_Rule: array (Rule range  0 ..  464) of Nonterminal := (-1,
-3,-5,-5,-7,-7,-8,-8,-8,
-8,-8,-8,-8,-8,-8,-8,-8,
-8,-9,-9,-9,-9,-19,-20,-10,
-10,-10,-25,-25,-28,-28,-28,-28,
-28,-28,-28,-11,-21,-37,-40,-40,
-40,-40,-36,-41,-45,-45,-30,-48,
-50,-50,-31,-32,-32,-42,-51,-43,
-53,-33,-33,-54,-23,-55,-57,-58,
-58,-34,-60,-60,-60,-63,-65,-29,
-66,-64,-69,-71,-71,-71,-71,-71,
-35,-26,-26,-73,-73,-77,-77,-77,
-80,-80,-80,-80,-80,-80,-80,-75,
-75,-81,-81,-81,-46,-46,-46,-46,
-46,-46,-68,-90,-87,-88,-91,-91,
-91,-91,-89,-92,-92,-92,-92,-44,
-93,-93,-93,-93,-6,-6,-6,-6,
-6,-100,-100,-47,-104,-106,-106,-106,
-107,-107,-107,-107,-107,-107,-111,-111,
-111,-111,-111,-111,-112,-112,-112,-113,
```

```
-113,-114,-114,-114,-114,-110,-109,-109,
-109,-119,-116,-117,-117,-117,-121,-121,
-121,-121,-121,-121,-121,-121,-121,-121,
-122,-122,-122,-122,-122,-122,-138,-123,
-124,-132,-139,-133,-143,-134,-147,-147,
-147,-147,-148,-135,-125,-126,-127,-12,
-155,-155,-157,-157,-86,-158,-159,-161,
-161,-161,-83,-13,-165,-84,-27,-27,
-79,-18,-18,-18,-18,-14,-170,-85,
-173,-173,-174,-136,-177,-128,-137,-137,
-137,-178,-181,-184,-184,-184,-185,-186,
-187,-179,-180,-129,-2,-194,-194,-196,
-196,-196,-196,-196,-197,-197,-198,-195,
-202,-82,-82,-82,-199,-16,-204,-205,
-205,-130,-15,-207,-207,-208,-210,-210,
-210,-210,-212,-212,-212,-212,-212,-212,
-17,-17,-17,-216,-217,-220,-220,-221,
-78,-78,-223,-223,-223,-225,-226,-227,
-231,-232,-224,-131,-61,-61,-4,-4,
-234,-234,-22,-22,-24,-24,-38,-38,
-39,-39,-49,-49,-52,-52,-56,-56,
-59,-59,-62,-62,-67,-67,-70,-70,
-72,-72,-74,-74,-76,-76,-94,-94,
-95,-95,-97,-97,-99,-99,-235,-235,
-96,-96,-98,-98,-101,-101,-102,-236,
-236,-103,-103,-103,-105,-105,-108,-108,
-115,-118,-118,-120,-120,-140,-140,-141,
-141,-142,-237,-237,-144,-144,-146,-146,
-145,-145,-149,-149,-150,-150,-151,-151,
-238,-238,-152,-152,-153,-153,-154,-154,
-156,-156,-160,-160,-162,-162,-163,-164,
-164,-166,-166,-167,-167,-168,-168,-169,
-169,-171,-171,-172,-172,-239,-239,-240,
-240,-175,-175,-176,-176,-182,-182,-183,
-183,-188,-189,-190,-191,-191,-192,-192,
-193,-193,-200,-201,-201,-241,-241,-203,
-203,-206,-206,-209,-209,-211,-211,-213,
-213,-213,-215,-214,-214,-218,-218,-219,
-219,-222,-228,-229,-229,-230,-230,-233);
end Ada_Goto;
```

```
-- ADA.A
--
with ada_lex;
use ada_lex;
package parser is

   procedure yyparse;

   error_string : string(1..80);
   error_length : integer := 0;
   echo : boolean := false;
   number_of_errors : natural := 0;

end parser;
with ada_tokens, ada_goto, ada_shift_reduce, ada_lex, text_io, direct_io;
use  ada_tokens, ada_goto, ada_shift_reduce, ada_lex, text_io;
package body parser is

   procedure yyerror is

     first,last : integer := 0;

   begin
        number_of_errors := number_of_errors + 1;
        ada_lex.error_line(1..10) := "  <<< *** ";
        ada_lex.error_line(11..22) := "Syntax Error";
      ada_lex.error_line(23..30):= " *** >>>";
   end yyerror;


procedure YYParse is

  -- Rename User Defined Packages to Internal Names.
   package yy_goto_tables         renames
     Ada_Goto;
   package yy_shift_reduce_tables renames
     Ada_Shift_Reduce;
   package yy_tokens              renames
     Ada_Tokens;

  use yy_tokens, yy_goto_tables, yy_shift_reduce_tables;

  procedure yyerrok;
  procedure yyclearin;

  package yy is

     -- the size of the value and state stacks
     stack_size : constant Natural := 300;

     -- subtype rule         is natural;
     subtype parse_state  is natural;
     -- subtype nonterminal  is integer;
```

```
  -- encryption constants
  default           : constant := -1;
  first_shift_entry : constant :=  0;
  accept_code       : constant := -1001;
  error_code        : constant := -1000;

  -- stack data used by the parser
  tos               : natural := 0;
  value_stack       : array(0..stack_size) of yy_tokens.yystype;
  state_stack       : array(0..stack_size) of parse_state;

  -- current input symbol and action the parser is on
  action            : integer;
  rule_id           : rule;
  input_symbol      : yy_tokens.token;


  -- error recovery flag
  error_flag : natural := 0;
    -- indicates  3 - (number of valid shifts after an error occurs)

  look_ahead : boolean := true;
  index      : integer;

  -- Is Debugging option on or off
   DEBUG : constant boolean := false;

end yy;


function goto_state
  (state : yy.parse_state;
   sym   : nonterminal) return yy.parse_state;

function parse_action
  (state : yy.parse_state;
   t     : yy_tokens.token) return integer;

pragma inline(goto_state, parse_action);


function goto_state(state : yy.parse_state;
                    sym   : nonterminal) return yy.parse_state is
   index : integer;
begin
   index := goto_offset(state);
   while  integer(goto_matrix(index).nonterm) /= sym loop
      index := index + 1;
   end loop;
   return integer(goto_matrix(index).newstate);
end goto_state;
```

```
   function parse_action(state : yy.parse_state;
                         t     : yy_tokens.token) return integer is
      index      : integer;
      tok_pos    : integer;
      default    : constant integer := -1;
   begin
      tok_pos := yy_tokens.token'pos(t);
      index   := shift_reduce_offset(state);
      while integer(shift_reduce_matrix(index).t) /= tok_pos and then
            integer(shift_reduce_matrix(index).t) /= default
      loop
         index := index + 1;
      end loop;
      return integer(shift_reduce_matrix(index).act);
   end parse_action;

-- error recovery stuff

   procedure handle_error is
     temp_action : integer;
   begin

     if yy.error_flag = 3 then -- no shift yet, clobber input.
     if yy.debug then
        put_line("Ayacc.YYParse: Error Recovery Clobbers " &
                 yy_tokens.token'image(yy.input_symbol));
     end if;
      if yy.input_symbol = yy_tokens.end_of_input then  -- don't discard,
      if yy.debug then
         put_line("Ayacc.YYParse: Can't discard END_OF_INPUT, quiting...");
      end if;
      raise syntax_error;
      end if;

         yy.look_ahead := true;   -- get next token
      return;                    -- and try again...
   end if;

   if yy.error_flag = 0 then -- brand new error
      yyerror;
   end if;

   yy.error_flag := 3;

   -- find state on stack where error is a valid shift --

   if yy.debug then
      put_line("Ayacc.YYParse: Looking for state with error as valid shift");
   end if;

   loop
      if yy.debug then
```

```
      put_line("Ayacc.YYParse: Examining State " &
          yy.parse_state'image(yy.state_stack(yy.tos)));
      end if;
      temp_action := parse_action(yy.state_stack(yy.tos), error);

        if temp_action >= yy.first_shift_entry then
          yy.tos := yy.tos + 1;
          yy.state_stack(yy.tos) := temp_action;
          exit;
        end if;

      Decrement_Stack_Pointer :
      begin
       yy.tos := yy.tos - 1;
      exception
       when Constraint_Error =>
        yy.tos := 0;
      end Decrement_Stack_Pointer;

      if yy.tos = 0 then
       if yy.debug then
        put_line("Ayacc.YYParse: Error recovery popped entire stack, aborting...");
       end if;
       raise syntax_error;
      end if;
    end loop;

    if yy.debug then
      put_line("Ayacc.YYParse: Shifted error token in state " &
          yy.parse_state'image(yy.state_stack(yy.tos)));
    end if;

    end handle_error;

   -- print debugging information for a shift operation
   procedure shift_debug(state_id: yy.parse_state; lexeme: yy_tokens.token) is
   begin
     put_line("Ayacc.YYParse: Shift "& yy.parse_state'image(state_id)&" on input
symbol "&
           yy_tokens.token'image(lexeme) );
   end;

   -- print debugging information for a reduce operation
   procedure reduce_debug(rule_id: rule; state_id: yy.parse_state) is
   begin
     put_line("Ayacc.YYParse: Reduce by rule "&rule'image(rule_id)&" goto state "&
           yy.parse_state'image(state_id));
   end;

   -- make the parser believe that 3 valid shifts have occured.
   -- used for error recovery.
   procedure yyerrok is
   begin
```

```
    yy.error_flag := 0;
 end yyerrok;

 -- called to clear input symbol that caused an error.
 procedure yyclearin is
 begin
    -- yy.input_symbol := yylex;
    yy.look_ahead := true;
 end yyclearin;

 procedure WRITE_HW is

   package HW_INOUT is new DIRECT_IO(TOKEN_NODE);
   use HW_INOUT;

 HW_FILE  : HW_INOUT.FILE_TYPE;
 HW_NODE  : TOKEN_NODE;
 REC_NO   : HW_INOUT.COUNT;

 begin
   OPEN(HW_FILE,OUT_FILE,"hw.dat");
   REC_NO := 1;
   current_ptr := hw_ptr.next;
   while current_ptr /= null loop
     HW_NODE := current_ptr.all;
     WRITE(HW_FILE,HW_NODE,REC_NO);
     current_ptr := current_ptr.next;
     REC_NO := REC_NO + 1;
   end loop;
    CLOSE(HW_FILE);
 end WRITE_HW;

begin
   -- initialize by pushing state 0 and getting the first input symbol
      yy.state_stack(yy.tos) := 0;
   --    yy.state_stack(yy.tos) := 360; -- for a  case statement

   loop

      yy.index := shift_reduce_offset(yy.state_stack(yy.tos));
      if integer(shift_reduce_matrix(yy.index).t) = yy.default then
         yy.action := integer(shift_reduce_matrix(yy.index).act);
      else
         if yy.look_ahead then
            yy.look_ahead   := false;
            yy.input_symbol := yylex;
            current_ptr.token_name := yy.input_symbol;
         end if;
         yy.action :=
          parse_action(yy.state_stack(yy.tos), yy.input_symbol);
      end if;

      if yy.action >= yy.first_shift_entry then  -- SHIFT
```

```
        if yy.debug then
           shift_debug(yy.action, yy.input_symbol);
        end if;

        -- Enter new state
        yy.tos := yy.tos + 1;
        yy.state_stack(yy.tos) := yy.action;
        yy.value_stack(yy.tos) := yylval;

     if yy.error_flag > 0 then  -- indicate a valid shift
        yy.error_flag := yy.error_flag - 1;
     end if;

        -- Advance lookahead
        yy.look_ahead := true;

     elsif yy.action = yy.error_code then       -- ERROR
        if yy.debug then
          put("error_code = ");
          put_line(integer'image(yy.error_code));
        end if;
        handle_error;

     elsif yy.action = yy.accept_code then
        if yy.debug then
           put_line("Ayacc.YYParse: Accepting Grammar...");
        end if;
        exit;

     else -- Reduce Action

        -- Convert action into a rule
        yy.rule_id  := -1 * yy.action;

        -- Execute User Action
        -- user_action(yy.rule_id);
  put(integer'image(yy.rule_id));
        case yy.rule_id is


when  21 =>
--#line  205
error_length := 18;
error_string(1..error_length) := "object declaration";
assign_association(yy.rule_id);

when  22 =>
--#line  210
error_length := 18;
error_string(1..error_length) := "number declaration";
assign_association(yy.rule_id);
```

```
when 26 =>
--#line 221
error_length := 16;
error_string(1..error_length) := "type declaration";
assign_association(yy.rule_id);

when 43 =>
--#line 258
error_length := 23;
error_string(1..error_length) := "derived type declaration";
assign_association(yy.rule_id);

when 47 =>
--#line 274
error_length := 27;
error_string(1..error_length) := "enumeration type definintion";
assign_association(yy.rule_id);

when 51 =>
--#line 284
error_length := 23;
error_string(1..error_length) := "integer type definition";
assign_association(yy.rule_id);

when 53 =>
--#line 289
error_length := 20;
error_string(1..error_length) := "real type definition";
assign_association(yy.rule_id);

when 74 =>
--#line 378
error_length := 12;
error_string(1..error_length) := "variant part";
assign_association(yy.rule_id);

when 75 .. 80 =>
--#line 391
error_length := 6;
error_string(1..error_length) := "choice";
assign_association(yy.rule_id);

when 124 =>
--#line 483
error_length := 21;
error_string(1..error_length) := "component association";
assign_association(yy.rule_id);

when 162 =>
--#line 554
 null;

when 185 =>
```

```
--#line 587
error_length := 20;
error_string(1..20) := "assignment statement";
assign_association(yy.rule_id);

when 186 =>
--#line 596
error_length := 12;
error_string(1..error_length) := "if statement";
assign_association(yy.rule_id);

when 187 =>
--#line 599
error_length := 9;
error_string(1..error_length) := "condition";
assign_association(yy.rule_id);

when 188 =>
--#line 607
error_length := 14;
error_string(1..14) := "case statement";
assign_association(yy.rule_id);

when 189 =>
--#line 612
error_length := 11;
error_string(1..11) := "alternative";
assign_association(yy.rule_id);

when 190 =>
--#line 621
error_length := 14;
error_string(1..error_length) := "loop statement";
assign_association(yy.rule_id);

when 194 =>
--#line 630
error_length := 16;
error_string(1..error_length) := "iteration scheme";
assign_association(yy.rule_id);

when 195 =>
--#line 635
error_length := 28;
error_string(1..error_length) := "loop parameter specification";
assign_association(yy.rule_id);

when 196 =>
--#line 645
error_length := 15;
error_string(1..error_length) := "block statement";
assign_association(yy.rule_id);
```

```
when 200 =>
--#line 664
error_length := 22;
error_string(1..error_length) := "subprogram declaration";
assign_association(yy.rule_id);

when 206 =>
--#line 680
error_length := 11;
error_string(1..error_length) := "formal part";
assign_association(yy.rule_id);

when 207 =>
--#line 684
error_length := 23;
error_string(1..error_length) := "parameter specification";
assign_association(yy.rule_id);

when 211 =>
--#line 699
error_length := 15;
error_string(1..error_length) := "subprogram body";
assign_association(yy.rule_id);

when 213 =>
--#line 709
error_length := 21;
error_string(1..error_length) := "package specification";
assign_association(yy.rule_id);

when 214 =>
--#line 716
error_length := 12;
error_string(1..error_length) := "package body";
assign_association(yy.rule_id);

when 216 =>
--#line 724
error_length := 24;
error_string(1..error_length) := "private type declaration";
assign_association(yy.rule_id);

when 217 =>
--#line 731
error_length := 10;
error_string(1..error_length) := "use clause";
assign_association(yy.rule_id);

when 223 =>
--#line 751
error_length := 16;
error_string(1..error_length) := "task specification";
assign_association(yy.rule_id);
```

```
when 224 =>
--#line 759
error_length := 9;
error_string(1..error_length) := "task body";
assign_association(yy.rule_id);

when 228 =>
--#line 776
error_length := 16;
error_string(1..error_length) := "accept statement";
assign_association(yy.rule_id);

when 230 =>
--#line 783
error_length := 15;
error_string(1..error_length) := "delay statement";
assign_association(yy.rule_id);

when 233 =>
--#line 788
error_length := 16;
error_string(1..error_length) := "select statement";
assign_association(yy.rule_id);

when 235 =>
--#line 801
error_length := 18;
error_string(1..error_length) := "select alternative";
assign_association(yy.rule_id);

when 238 =>
--#line 805
error_length := 26;
error_string(1..error_length) := "selective wait alternative";
assign_association(yy.rule_id);

when 243 =>
--#line 837
error_length := 16;
error_string(1..error_length) := "timed entry call";
assign_association(yy.rule_id);

when 247 =>
--#line 851
error_length := 16;
error_string(1..error_length) := "compilation unit";
assign_association(yy.rule_id);

when 257 =>
--#line 873
error_length := 11;
error_string(1..error_length) := "with clause";
```

```
assign_association(yy.rule_id);

when 261 =>
--#line 885
error_length := 7;
error_string(1..error_length) := "subunit";
assign_association(yy.rule_id);

when 262 =>
--#line 891
error_length := 21;
error_string(1..error_length) := "exception declaration";
assign_association(yy.rule_id);

when 263 =>
--#line 897
error_length := 17;
error_string(1..error_length) := "exception handler";
assign_association(yy.rule_id);

when 265 =>
--#line 901
error_length := 16;
error_string(1..error_length) := "exception choice";
assign_association(yy.rule_id);

when 266 =>
--#line 905
error_length := 15;
error_string(1..error_length) := "raise statement";
assign_association(yy.rule_id);

when 269 =>
--#line 914
error_length := 21;
error_string(1..error_length) := "generic specification";
assign_association(yy.rule_id);

when 270 =>
--#line 918
error_length := 19;
error_string(1..error_length) := "generic formal part";
assign_association(yy.rule_id);

when 274 =>
--#line 926
error_length := 29;
error_string(1..error_length) := "generic parameter declaration";
assign_association(yy.rule_id);

when 280 =>
--#line 932
error_length := 23;
```

```
error_string(1..error_length) := "generic type definition";
assign_association(yy.rule_id);

when  283 =>
--#line  943
error_length := 18;
error_string(1..error_length) := "generic instantiation";
assign_association(yy.rule_id);

when  284 =>
--#line  948
error_length := 19;
error_string(1..error_length) := "generic actual part";
assign_association(yy.rule_id);

when  330 =>
--#line  1072
error_length := 9;
error_string(1..error_length) := "or choice";
assign_association(yy.rule_id);

when  437 =>
--#line  1327
error_length := 10;
error_string(1..error_length) := "use clause";
assign_association(yy.rule_id);

            when others => null;
         end case;

         -- Pop RHS states and goto next state
         yy.tos      := yy.tos - rule_length(yy.rule_id) + 1;
         if yy.tos = 0 then
           exit;
         end if;
         yy.state_stack(yy.tos) := goto_state(yy.state_stack(yy.tos-1) ,
                          get_lhs_rule(yy.rule_id));

         yy.value_stack(yy.tos) := yyval;

         if yy.debug then
            reduce_debug(yy.rule_id,
               goto_state(yy.state_stack(yy.tos - 1),
                        get_lhs_rule(yy.rule_id)));
         end if;

      end if;
   end loop;

 add_node(current_ptr);
current_ptr.external_variable := 1000;
current_ptr.association := current_ptr.previous.association;
new_line; linenum;
```

```
put(buffer_line);
buffer_line := empty_buffer;
buffer_length := 1;
if error_line /= empty_line then
  new_line;
  put(error_line);
  error_line := empty_line;
end if;

   WRITE_HW;
     new_line;

exception
  when Syntax_Error =>
    yyerror;

end yyparse;

end parser;
```

```
-- DIAGNOSE_PKG.A
--
with text_io, ada_lex, parser, ada_tokens;
use text_io, ada_lex, parser, ada_tokens;

package diagnosis is

  type list_array is array(integer range <>) of token_node;

  student_count,
  expert_count : integer := 0;
  list_count : integer := 0;
  list_ptr : token_ptr;
  hw_array,
  expert_array,
  student_array : list_array(1..50);
  student_ptr : token_ptr;
  expert_ptr : token_ptr;

  procedure reverse_list(list_ptr:in out token_ptr);
  procedure load_array(list_ptr:in token_ptr;
            list_count:in out integer; hw_array:in out list_array);
  procedure compare(expert_array,student_array:in list_array;
          expert_count,student_count:in integer);
  procedure diagnose(student_ptr, expert_ptr:in out token_ptr);

end diagnosis;

package body diagnosis is


  procedure reverse_list(list_ptr:in out token_ptr) is

  begin
     current_ptr := list_ptr;
     while current_ptr.next /= null loop
        if current_ptr.external_variable /= 0 then -- association
          current_ptr.next.token_name
             := current_ptr.token_name;
        end if;
       current_ptr := current_ptr.next;
     end loop;
     list_ptr := current_ptr; -- now at the end
  end reverse_list;

  procedure load_array(list_ptr:in token_ptr;
            list_count:in out integer; hw_array:in out list_array) is

    current_ptr,
    array_ptr : token_ptr := list_ptr;

  begin
     list_count := 0;
```

```
     current_ptr := list_ptr;
     array_ptr := list_ptr;
     while current_ptr.previous /= null loop -- not START
         if current_ptr.external_variable /= 0 then -- ass node
            list_count := list_count + 1;
            hw_array(list_count).next := new token_node;
            array_ptr := hw_array(list_count).next;
            array_ptr.all := current_ptr.all;
            array_ptr.next := null;
            array_ptr.previous := null;
         else       -- token node
             array_ptr.next := new token_node;
             array_ptr.next.all := current_ptr.all;
             array_ptr.next.next := null;
             array_ptr.next.previous := array_ptr;
             array_ptr := array_ptr.next;
        end if;
        current_ptr := current_ptr.previous;
    end loop;
 end load_array;

 procedure compare(expert_array,student_array:in list_array;
         expert_count, student_count: in integer) is

   subtype STATUS_TYPE is INTEGER range 0..2;
    STATUS,
    PREVIOUS_STATUS : STATUS_TYPE := 0;
    current_student_ptr : token_ptr;
   current_expert_ptr : token_ptr;
   found : boolean := false;
   CONSTRUCT, PREVIOUS_CONSTRUCT : integer;
   error_ptr,
   previous_error_ptr  : token_ptr := null;
    INIT : BOOLEAN := TRUE;

procedure PRINT_OUT(PREVIOUS_ERROR_PTR:in TOKEN_PTR) is

begin_line : boolean := false;
counter : integer := 0;
tab : integer := 7;
select_block : boolean := false;
subtype_line : boolean := false;
when_line : boolean := false;
end_line : boolean := false;
proc_line : boolean := false;
previous_token : token := END_OF_INPUT;
PRINT_PTR : TOKEN_PTR := PREVIOUS_ERROR_PTR;

  procedure SPACES(TAB:in INTEGER) is

  begin
    for I in 1 .. TAB loop
      PUT(" ");
```

```
    end loop;
  end SPACES;

begin
    CURRENT_PTR := PREVIOUS_ERROR_PTR;
    while CURRENT_PTR.NEXT /= null loop
      CURRENT_PTR := CURRENT_PTR.NEXT;
    end loop;
  if PREVIOUS_STATUS = 1 then
    new_line; PUT("*** MISSING ");
  else
    new_line; PUT("*** UNNECESSARY ");
  end if;

 PUT(PREVIOUS_ERROR_PTR.PREVIOUS.VALUE(1..
   PREVIOUS_ERROR_PTR.PREVIOUS.LENGTH));
 PUT(" ***"); NEW_LINE; SET_COL(8);
 PRINT_PTR := CURRENT_PTR;
 while PRINT_PTR /= null loop
   if PRINT_PTR.external_variable = 0 then -- a token node
     if 77 < (counter + PRINT_PTR.length) then
       begin_line := true;
     end if;
   if not INIT then
     case PRINT_PTR.token_name is
       when ')' | ',' | ';' | '.' | ''' => -- no space before
         null;
       when '(' =>
         if previous_token /= IDENTIFIER and previous_token /=
           '(' then
           put(" ");
         end if;
       when others =>
         if previous_token = '.' or previous_token = ''' or
           previous_token = '(' then
           null;
         elsif counter /= 0 then -- not at beginning of a line
             put(" ");
             counter := counter + 1;
         end if;
     end case;
     case PRINT_PTR.token_name is -- for tab before
       when ELSE_TOKEN | ELSIF_TOKEN | PRIVATE_TOKEN |
           EXCEPTION_TOKEN =>
         begin_line := true;
         tab := tab - 3;
       when END_TOKEN =>
         begin_line := true;
         tab := tab - 3;
       when OR_TOKEN =>
         if select_block then
           begin_line := true;
           tab := tab - 3;
```

```
         end if;
      when PROCEDURE_TOKEN | PACKAGE_TOKEN | FUNCTION_TOKEN =>
         if not proc_line then
            new_line;
         end if;
         begin_line := true;
         proc_line := true;
      when WITH_TOKEN =>
         new_line(2);
      when BEGIN_TOKEN =>
         if previous_token /= IS_TOKEN then
            new_line;
         end if;
         tab := tab - 3;
         begin_line := true;
      when SELECT_TOKEN =>
         if select_block then
            begin_line := true;
         end if;
      when WHEN_TOKEN =>
         tab := tab - 3; when_line := true;
      when others => null;
   end case;
end if;
   if begin_line then
      if (PRINT_PTR.token_name = ')' or
         PRINT_PTR.token_name = '(') and not when_line then
         begin_line := true;
      else
         new_line;
         spaces(tab);
         begin_line := false;
         counter := tab;
      end if;
   end if;
   put(PRINT_PTR.value(1..PRINT_PTR.length));
   INIT := FALSE;
   counter := counter + PRINT_PTR.length;
   case PRINT_PTR.token_name is -- tab after
      when BEGIN_TOKEN | RECORD_TOKEN | THEN_TOKEN | ELSE_TOKEN |
           PRIVATE_TOKEN | EXCEPTION_TOKEN =>
         tab := tab + 3;
         begin_line := true;
      when LOOP_TOKEN =>
         if previous_token /= END_TOKEN then -- loop statement
            begin_line := true;
            tab := tab + 3;
         end if;
      when '(' =>
         tab := tab + 3;
      when ')' =>
         tab := tab - 3;
      when ARROW =>
```

```
        if when_line then
          begin_line := true;
          tab := tab + 3;  -- case alternatives
        end if;
      when OR_TOKEN =>
        if select_block then
          tab := tab + 3;
          begin_line := true;
        end if;
      when SELECT_TOKEN =>
        if select_block then
          select_block := false;
        else
          select_block := true;
        end if;
      when CASE_TOKEN =>
        if previous_token = END_TOKEN then
          tab := tab - 3; -- end case
        else
          tab := tab + 3; -- case statement
        end if;
      when IDENTIFIER =>
        if previous_token = END_TOKEN then
          proc_line := false;
        end if;
      when ';' =>
        subtype_line := false;
        begin_line := true;
      when IS_TOKEN =>
        case PRINT_PTR.previous.token_name is
          when NEW_TOKEN | ARRAY_TOKEN | ACCESS_TOKEN |
               DIGITS_TOKEN | DELTA_TOKEN | SEPARATE_TOKEN =>
            null;
          when others =>
            if not subtype_line then
              tab := tab + 3;
              begin_line := true;
              if proc_line then
                new_line;
                proc_line := false;
              end if;
            end if;
        end case;
      when SUBTYPE_TOKEN =>
        subtype_line := true;
      when WHEN_TOKEN =>
        when_line := true;
      when others => null;
    end case;
    previous_token := PRINT_PTR.token_name;
  end if;
     PRINT_PTR := PRINT_PTR.previous;
end loop;
```

```
end PRINT_OUT;

procedure ERROR(construct:in integer;error_ptr:in TOKEN_PTR;
    STATUS:in STATUS_TYPE) is

 CURRENT_PTR,
 FIRST_LIST,
 LAST_LIST,
 FIRST_PREVIOUS_LIST,
 LAST_PREVIOUS_LIST   : TOKEN_PTR;

 begin
   FIRST_LIST := ERROR_PTR.NEXT;
    CURRENT_PTR := FIRST_LIST;
    while CURRENT_PTR /= null loop
      CURRENT_PTR := CURRENT_PTR.NEXT;
    end loop;
    LAST_LIST := CURRENT_PTR;
    if STATUS = PREVIOUS_STATUS then
      CURRENT_PTR := PREVIOUS_ERROR_PTR;
      while CURRENT_PTR.NEXT /= null loop
        CURRENT_PTR := CURRENT_PTR.NEXT;
      end loop;
      LAST_PREVIOUS_LIST := CURRENT_PTR;
      if LAST_PREVIOUS_LIST.association =
        FIRST_LIST.association + 1 then -- consecutive
        LAST_PREVIOUS_LIST.NEXT := FIRST_LIST;
        FIRST_LIST.PREVIOUS := LAST_PREVIOUS_LIST;
       CURRENT_PTR := PREVIOUS_ERROR_PTR;
     else
        PREVIOUS_CONSTRUCT := CONSTRUCT;
     end if;

   elsif PREVIOUS_STATUS = 0 then  -- first error
     PREVIOUS_ERROR_PTR := ERROR_PTR.NEXT;
     PREVIOUS_CONSTRUCT := CONSTRUCT;
  else
    PREVIOUS_ERROR_PTR := FIRST_LIST;
    PREVIOUS_CONSTRUCT := CONSTRUCT;
    PRINT_OUT(PREVIOUS_ERROR_PTR);
    PREVIOUS_ERROR_PTR := null; -- status change
    PREVIOUS_CONSTRUCT := 0;
 end if;
 PREVIOUS_STATUS := STATUS;

 end ERROR;

begin
   for i in 1 .. expert_count loop
     for j in 1 .. student_count loop
        current_expert_ptr := expert_array(i).next;
        current_student_ptr := student_array(j).next;
        if current_expert_ptr.external_variable > 0 and
```

```
        (current_expert_ptr.external_variable =
        current_student_ptr.external_variable) then  -- same construct
        current_student_ptr := current_student_ptr.next;
        current_expert_ptr := current_expert_ptr.next;
        while current_expert_ptr /= null and
           current_student_ptr /= null loop
          if current_student_ptr.value =
            current_expert_ptr.value then
            found := true;
          else
            found := false;
          exit;
          end if;
          current_student_ptr := current_student_ptr.next;
          current_expert_ptr := current_expert_ptr.next;
        end loop;
        if found then
           exit;
        end if;
      end if;
    end loop;
     if not found then
      construct := expert_array(i).next.external_variable;
      STATUS := 2;
      ERROR(construct,expert_array(i).next,STATUS);
     else
      found := false;
     end if;
  end loop;
  for i in 1 .. student_count loop
    for j in 1 .. expert_count loop
      current_expert_ptr := expert_array(j).next;
      current_student_ptr := student_array(i).next;
     if current_student_ptr.external_variable > 0 and
        (current_expert_ptr.external_variable =
        current_student_ptr.external_variable) then  -- same construct
        current_student_ptr := current_student_ptr.next;
        current_expert_ptr := current_expert_ptr.next;
        while current_expert_ptr /= null and
           current_student_ptr /= null loop
          if current_student_ptr.value =
            current_expert_ptr.value then
            found := true;
          else
            found := false;
          exit;
          end if;
          current_student_ptr := current_student_ptr.next;
          current_expert_ptr := current_expert_ptr.next;
        end loop;
        if found then
           exit;
        end if;
```

```
        end if;
      end loop;
       if not found then
         construct := student_array(i).next.external_variable;
         STATUS := 2;
         ERROR(construct,student_array(i).next,STATUS);
       else
         found := false;
       end if;
    end loop;
    if PREVIOUS_STATUS /= 0 then -- print out last error
      PRINT_OUT(PREVIOUS_ERROR_PTR);
     else
       PUT("NO ERRORS");
    end if;
  end compare;

  procedure diagnose(student_ptr, expert_ptr:in out token_ptr) is

  begin
      reverse_list(student_ptr);
      load_array(student_ptr,student_count,student_array);
      reverse_list(expert_ptr);
      load_array(expert_ptr,expert_count,expert_array);
      compare(expert_array,student_array,expert_count,
         student_count);
  end diagnose;

end diagnosis;
```

-- EDITOR.A

```
--
package EDITOR_PKG is

  type LINE_ARRAY is array(1..49) of STRING(1..78);
  EMPTY_STRING          : constant STRING(1..78) := (others => ' ');
  LINES                 : LINE_ARRAY := (others => EMPTY_STRING);
  CHECK                 : BOOLEAN := FALSE;
  LAST_LINE             : INTEGER := 0;

  procedure EDITOR;

end EDITOR_PKG;

with TEXT_IO;
use TEXT_IO;
package body EDITOR_PKG is

  package INTEGER_INOUT is new INTEGER_IO(INTEGER);
  use INTEGER_INOUT;

  procedure ERROR is
  begin
    PUT_LINE("THIS IS AN ERROR");
  end ERROR;

  procedure ADD(command:in STRING;LINES:in out LINE_ARRAY) is

  DATA_LINE : STRING(1..78) := EMPTY_STRING;
  START_LINE,STOP_LINE : INTEGER;
  CHOICE3,CHOICE4 : CHARACTER;
  END_OF_INPUT : BOOLEAN := FALSE;
  DATA_LENGTH : INTEGER;

  begin
    CHOICE3 := COMMAND(3);
    case CHOICE3 is
      when '0' .. '9' => -- >= 1 digit, choice3
        start_line := CHARACTER'POS(CHOICE3)-48;
        CHOICE4 := COMMAND(4);
        case CHOICE4 is
          when '0'..'9' => -- 2 digits, choice4
            start_line := ((CHARACTER'POS(CHOICE3)-48) * 10) +
                     (CHARACTER'POS(CHOICE4)-48);
            while not END_OF_INPUT loop
              INTEGER_INOUT.PUT(start_line,width => 2);
              PUT(" ");
              get_line(DATA_LINE,DATA_LENGTH);
              if DATA_LINE(1) = '.' then
                END_OF_INPUT := TRUE;
                exit;
              end if;
              for I in reverse start_line  .. 48 loop
```

```
            LINES(I + 1) := LINES(I);
          end loop;
          LINES(start_line) := DATA_LINE;
          last_line := last_line + 1;
          start_line := start_line + 1;
          DATA_LINE := EMPTY_STRING;
        end loop;
      when ' ' =>  -- choice4
        while not END_OF_INPUT loop
          INTEGER_INOUT.PUT(start_line,width => 2);
          PUT(" ");
          get_line(DATA_LINE,DATA_LENGTH);
          if DATA_LINE(1) = '.' then
            END_OF_INPUT := TRUE;
            exit;
          end if;
          for I in reverse start_line  .. 48 loop
            LINES(I + 1) := LINES(I);
          end loop;
          LINES(start_line)(1..DATA_LENGTH)
            := DATA_LINE(1..DATA_LENGTH);
          LINES(start_line)(DATA_LENGTH + 1)
            := ASCII.CR;
          last_line := last_line + 1;
          start_line := start_line + 1;
          DATA_LINE := EMPTY_STRING;
        end loop;
      when others => error; -- choice 4
    end case;
  when others => error; -- choice 3
 end case;
end ADD;

procedure DELETE(command:in STRING;LINES:in out LINE_ARRAY) is

START_LINE,STOP_LINE : INTEGER;
COUNTER : INTEGER;
CHOICE3,CHOICE4,CHOICE5,CHOICE6,CHOICE7 : CHARACTER;

begin
 CHOICE3 := COMMAND(3);
 case CHOICE3 is
  when 'a' | 'A' =>
    for I in 1..last_line loop
     LINES(I) := EMPTY_STRING;
    end loop;
    last_line := 0;
  when '0' .. '9' => -- >= 1 digit, choice3
    start_line := CHARACTER'POS(CHOICE3)-48;
    CHOICE4 := COMMAND(4);
    case CHOICE4 is
      when '0'..'9' => -- 2 digits, choice4
        start_line := ((CHARACTER'POS(CHOICE3)-48) * 10) +
```

```
          (CHARACTER'POS(CHOICE4)-48);
  CHOICE5 := COMMAND(5);
  case CHOICE5 is
    when '-' | '/' => -- delimeter for 2 digit range, choice5
      CHOICE6 := COMMAND(6);
      case CHOICE6 is
        when '0'..'9' => -- >= 1 digit stop, choice6
          CHOICE7 := COMMAND(7);
          case CHOICE7 is
            when '0' .. '9' => -- 2 digit stop,choice7
              stop_line := ((CHARACTER'POS(CHOICE6)-48)
                * 10) + (CHARACTER'POS(CHOICE7)-48);
              counter := start_line;
              for I in stop_line + 1 .. last_line loop
                LINES(counter) := LINES(I);
                counter := counter + 1;
              end loop;
              for I in 1 .. ((stop_line - start_line)
               + 1) loop
                LINES(last_line) := EMPTY_STRING;
                last_line := last_line - 1;
              end loop;
            when others => error; -- choice7
          end case;
        when others => error;  -- choice6
      end case;
    when ' ' => -- one line, choice5
      for I in start_line  .. last_line - 1 loop
        LINES(I) := LINES(I + 1);
      end loop;
      LINES(last_line) := EMPTY_STRING;
    when others => error; -- choice5
  end case;
when '-' | '/' => -- delimeter for 1 digit range,choice4
  CHOICE5 := COMMAND(5);
  case CHOICE5 is
    when '0'..'9' => -- >= 1 digit stop,choice5
      stop_line := CHARACTER'POS(CHOICE5)-48;
      CHOICE6 := COMMAND(6);
      case CHOICE6 is
        when '0'..'9'  => -- 2 digit stop,choice6
          stop_line := ((CHARACTER'POS(CHOICE5)-48) * 10) +
                  (CHARACTER'POS(CHOICE6)-48);
          counter := start_line;
          for I in stop_line + 1 .. last_line loop
            LINES(counter) := LINES(I);
            counter := counter + 1;
          end loop;
          for I in 1 .. ((stop_line - start_line) + 1) loop
            LINES(last_line) := EMPTY_STRING;
            last_line := last_line - 1;
          end loop;
        when ' ' => -- 1 digit range
```

```
                    counter := start_line;
                    for I in stop_line + 1 .. last_line loop
                      LINES(counter) := LINES(I);
                      counter := counter + 1;
                    end loop;
                    for I in 1 ..((stop_line - start_line) + 1) loop
                      LINES(last_line) := EMPTY_STRING;
                      last_line := last_line - 1;
                    end loop;
                  when others => error;
                end case;
              when others => error; -- choice5
            end case;
          when ' ' =>  -- choice4
            for I in start_line .. last_line - 1 loop
              LINES(I) := LINES(I + 1);
            end loop;
            LINES(last_line) := EMPTY_STRING;
          when others => error; -- choice 4
        end case;
      when others => error; -- choice 3
    end case;
end DELETE;

procedure LIST(command:in STRING;LINES:in LINE_ARRAY) is

LINE : STRING(1..78) := EMPTY_STRING;
START_LINE,STOP_LINE : INTEGER;
CHOICE3,CHOICE4,CHOICE5,CHOICE6,CHOICE7 : CHARACTER;

begin
  CHOICE3 := COMMAND(3);
  case CHOICE3 is
    when 'a' | 'A' =>
      put(character'val(12)); -- clear screen
      for I in 1 .. last_line loop
        LINE := LINES(I);
        if I = 25 then
          put("        . . . more");
          SKIP_LINE;
          put(character'val(12));
        end if;
        INTEGER_INOUT.PUT(I,WIDTH => 2);
        PUT(" ");
        PUT_LINE(LINE);
      end loop;
    when '0' .. '9' => -- >= 1 digit, choice3
      start_line := CHARACTER'POS(CHOICE3)-48;
      CHOICE4 := COMMAND(4);
      case CHOICE4 is
        when '0'..'9' => -- 2 digits, choice4
          start_line := ((CHARACTER'POS(CHOICE3)-48) * 10) +
                    (CHARACTER'POS(CHOICE4)-48);
```

```
        CHOICE5 := COMMAND(5);
        case CHOICE5 is
          when '-' | '/' => -- delimeter for 2 digit range, choice5
            CHOICE6 := COMMAND(6);
            case CHOICE6 is
              when '0'..'9' => -- >= 1 digit stop, choice6
                CHOICE7 := COMMAND(7);
                case CHOICE7 is
                  when '0' .. '9' => -- 2 digit stop,choice7
                    stop_line := ((CHARACTER'POS(CHOICE6)-48)
                      * 10) + (CHARACTER'POS(CHOICE7)-48);
                    NEW_LINE;
                    for I in start_line .. stop_line loop
                      LINE := LINES(I);
                      INTEGER_INOUT.PUT(I,width => 2);
                      PUT(" ");
                      PUT_LINE(LINE);
                    end loop;
                  when others => error; -- choice7
                end case;
              when others => error; -- choice6
            end case;
          when ' ' => -- one line, choice5
            NEW_LINE;
            LINE := LINES(start_line);
            INTEGER_INOUT.PUT(start_line,width => 2);
            PUT(" ");
            PUT_LINE(LINE);
          when others => error; -- choice5
        end case;
      when '-' | '/' => -- delimeter for 1 digit range,choice4
        CHOICE5 := COMMAND(5);
        case CHOICE5 is
          when '0'..'9' => -- >= 1 digit stop,choice5
            stop_line := CHARACTER'POS(CHOICE5)-48;
            CHOICE6 := COMMAND(6);
            case CHOICE6 is
              when '0'..'9'  => -- 2 digit stop,choice6
                stop_line := ((CHARACTER'POS(CHOICE5)-48) * 10) +
                         (CHARACTER'POS(CHOICE6)-48);
                NEW_LINE;
                for I in start_line .. stop_line loop
                  LINE := LINES(I);
                  INTEGER_INOUT.PUT(I,width => 2);
                  PUT(" ");
                  PUT_LINE(LINE);
                end loop;
              when ' ' => -- 1 digit range
               NEW_LINE;
               for I in start_line .. stop_line loop
                 LINE := LINES(I);
                 INTEGER_INOUT.PUT(I,width => 2);
                 PUT(" ");
```

```
                PUT_LINE(LINE);
              end loop;
             when others => error;
           end case;
         when others => error; -- choice5
       end case;
     when ' ' =>  -- choice4
       NEW_LINE;
       LINE := LINES(start_line);
       INTEGER_INOUT.PUT(start_line,width => 2);
       PUT(" ");
       PUT_LINE(LINE);
     when others => error; -- choice 4
    end case;
   when others => error; -- choice 3
  end case;
end LIST;

procedure MODIFY(LINE:in out STRING; LINE_NUM:in INTEGER) is

C : CHARACTER;
CHANGE : BOOLEAN := FALSE;
PLACE : INTEGER := 0;

begin
 loop
  INTEGER_INOUT.PUT(LINE_NUM,width => 2);
  PUT(" ");
  PUT_LINE(LINE);
  PUT("   ");
   while not END_OF_LINE loop
    GET(C);
    if CHANGE then
      for I in reverse PLACE .. 74 loop
        LINE(I + 1) := LINE(I);
      end loop;
      LINE(PLACE) := C;
      PLACE := PLACE + 1;
     elsif C = 'd' or C = 'D' then
      for I in PLACE + 1 .. 74 loop
        LINE(I) := LINE(I + 1);
      end loop;
      LINE(78) := ' ';
     elsif C = 'i' or c = 'I' then
      CHANGE := TRUE;
      PLACE := PLACE + 1;
     else
      PLACE := PLACE + 1;
     end if;
  end loop;
  CHANGE := FALSE;
  SKIP_LINE;
  if PLACE = 0 then
```

```
    exit;
  end if;
  PLACE := 0;
end loop;
end MODIFY;

procedure MODIFY(command:in STRING;LINES:in out LINE_ARRAY) is

LINE : STRING(1..78);
START_LINE,STOP_LINE : INTEGER;
CHOICE3,CHOICE4,CHOICE5,CHOICE6,CHOICE7 : CHARACTER;

begin
  CHOICE3 := COMMAND(3);
  case CHOICE3 is
    when '0' .. '9' => -- >= 1 digit, choice3
      start_line := CHARACTER'POS(CHOICE3)-48;
      CHOICE4 := COMMAND(4);
      case CHOICE4 is
        when '0'..'9' => -- 2 digits, choice4
          start_line := ((CHARACTER'POS(CHOICE3)-48) * 10) +
                  (CHARACTER'POS(CHOICE4)-48);
          CHOICE5 := COMMAND(5);
          case CHOICE5 is
            when '-' | '/' => -- delimeter for 2 digit range, choice5
              CHOICE6 := COMMAND(6);
              case CHOICE6 is
                when '0'..'9' => -- >= 1 digit stop, choice6
                  CHOICE7 := COMMAND(7);
                  case CHOICE7 is
                    when '0' .. '9' => -- 2 digit stop,choice7
                      stop_line := ((CHARACTER'POS(CHOICE6)-48)
                        * 10) + (CHARACTER'POS(CHOICE7)-48);
                      NEW_LINE;
                      for I in start_line .. stop_line loop
                        LINE := LINES(I);
                        MODIFY(LINE,I);
                        LINES(I) := LINE;
                      end loop;
                    when others => error; -- choice7
                  end case;
                when others => error; -- choice6
              end case;
            when ' ' => -- one line, choice5
              NEW_LINE;
              LINE := LINES(start_line);
              MODIFY(LINE,start_line);
              LINES(start_line) := LINE;
            when others => error; -- choice5
          end case;
        when '-' | '/' => -- delimeter for 1 digit range,choice4
          CHOICE5 := COMMAND(5);
          case CHOICE5 is
```

```
              when '0'..'9' => -- >= 1 digit stop,choice5
                stop_line := CHARACTER'POS(CHOICE5)-48;
                CHOICE6 := COMMAND(6);
                case CHOICE6 is
                  when '0'..'9'  => -- 2 digit stop,choice6
                    stop_line := ((CHARACTER'POS(CHOICE5)-48) * 10) +
                                 (CHARACTER'POS(CHOICE6)-48);
                    NEW_LINE;
                    for I in start_line .. stop_line loop
                      LINE := LINES(I);
                      MODIFY(LINE,I);
                      LINES(I) := LINE;
                    end loop;
                  when ' ' => -- 1 digit range
                   NEW_LINE;
                   for I in start_line .. stop_line loop
                     LINE := LINES(I);
                     MODIFY(LINE,I);
                     LINES(I) := LINE;
                   end loop;
                  when others => error;
                end case;
              when others => error; -- choice5
            end case;
          when ' ' =>  -- choice4
            NEW_LINE;
            LINE := LINES(start_line);
            MODIFY(LINE,start_line);
            LINES(start_line) := LINE;
          when others => error; -- choice 4
        end case;
      when others => error; -- choice 3
    end case;
  end MODIFY;

procedure EDITOR is

  CHOICE                : CHARACTER;
  COMMAND                 : STRING(1..78) := (others => ' ');
  COMMAND_LENGTH          : NATURAL := 0;

begin
  LINES := (others => EMPTY_STRING);
  LAST_LINE := 0;
  loop
    COMMAND := EMPTY_STRING;
    PUT("  @ ");
    TEXT_IO.GET_LINE(COMMAND,COMMAND_LENGTH);
    CHOICE := COMMAND(1);
    case CHOICE is
      when 'A' | 'a' =>
        ADD(COMMAND,LINES);
      when 'M' | 'm' =>
```

```
      MODIFY(COMMAND,LINES);
    when 'L' | 'l' =>
      LIST(COMMAND,LINES);
    when 'D' | 'd' =>
      DELETE(COMMAND,LINES);
    when 'Q' | 'q' =>
      CHECK := FALSE;
      exit;
    when 'C' | 'c' =>
      CHECK := TRUE;
      exit;
    when '?' =>
      new_line(2);
      put_line("(P)roblem statement     p       -- restates the problem");
      put_line("(C)heck                 c       -- checks the response");           put_line("(Q)uit
q       -- quits the editor");
      new_line;
      put_line("(A)dd line              a 8     -- add lines after line 8");
      put_line("(D)elete line #/range   d 8     -- removes line 8 from sequence and
renumbers");
      put_line("(L)ist all              l all   -- lists all the lines in the file");
      put_line("(L)ist line #/range     l 8-12  -- lists lines 8 through 12, inclusive");
      new_line;
      put_line("(M)odify line #/range   m 8     -- prints out line 8 and enables
insertions/");
      put_line("                                   deletions");
      new_line;
      put_line("                        8 The brown fox jumped over the lazy dog.");
      put_line("(I)nsert                    i quick ");
      put_line("                        8 The quick brown fox jumped over the lazy dog.");
      put_line("(D)elete                                            ddddd ");
      new_line(2);
    when others =>
      PUT_LINE("THAT IS NOT A VALID OPTION");
  end case;
 end loop;
end EDITOR;

end EDITOR_PKG;
```

```
-- INSTRUCTION.A
--
package SCREENS_PKG is

  subtype SCREEN_LENGTH is INTEGER range 1..14; -- screens in the buffer
  subtype LINE_LENGTH is INTEGER range 1..25; -- lines per screen
  type SCREEN_ARRAY is array(SCREEN_LENGTH,LINE_LENGTH) of
STRING(1..80);
  SCREEN_NUMBER : INTEGER := 1;
  SCREEN_COUNT  : INTEGER := 1;
  SCREEN_DATA   : SCREEN_ARRAY;
  CHECK         : BOOLEAN := FALSE;

  type TOPIC_SCREENS is array(1..12) of INTEGER;
  TOPIC_SCREENS_ARRAY : TOPIC_SCREENS := (5, 13, 22, 28, 33, 41, 48,
                             63, 75, 81, 89, 103);
  type TOPIC_PROB is array(1..12) of INTEGER;
  subtype TOPICS is INTEGER range 1..12;
  DUMB_STUDENT : constant TOPIC_PROB := (others => 0);
  type USER_RECORD is
    record
      USER_NAME : STRING(1..10) := "          ";
      COMPLETED_TOPIC : TOPICS := 1;
      USER_TOPICS : TOPIC_PROB := DUMB_STUDENT;
    end record;

  CURRENT_TOPIC_NUMBER : TOPICS := 1;
  USER_ID      : STRING(1..10);
  USER_ID_LENGTH : INTEGER;
  USER_NUMBER : INTEGER;
  USER_SPECIFIC : BOOLEAN := FALSE;

  subtype PROBLEM_TYPE is INTEGER range 1..5;
  type PROBLEM_INFO is
    record
      START_STATE  : INTEGER;
      PROB_START   : INTEGER;
      PROB_STOP    : INTEGER;
      SOLN_START   : INTEGER;
      SOLN_STOP    : INTEGER;
    end record;
  type TOPIC_PROBLEMS is array(1..5) of PROBLEM_INFO;
  type PROBLEM_DATA is array(1..12) of TOPIC_PROBLEMS;

  ITS_ADA_DATA  : constant PROBLEM_DATA := (((1,1,3,1,4),
                    (1,1,3,1,4),
                    (6,7,8,9,10),
                    (6,7,8,9,10),
                    (6,7,8,9,10)),
                    ((1,2,3,4,5),
                    (6,7,8,9,10),
                    (6,7,8,9,10),
                    (6,7,8,9,10),
```

(6,7,8,9,10)),
((1,2,3,4,5),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10)),
((1,2,3,4,5),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10)),
((1,2,3,4,5),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10)),
((1,2,3,4,5),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10)),
((1,2,3,4,5),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10)),
((1,2,3,4,5),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10)),
((1,2,3,4,5),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10)),
((1,2,3,4,5),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10)),
((1,2,3,4,5),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10)),
((1,2,3,4,5),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10),
(6,7,8,9,10)));

```
  procedure QUIT_PROGRAM;
  procedure LOAD_SCREENS(SCREEN_NUMBER:in INTEGER;
        SCREEN_DATA:in out SCREEN_ARRAY);
  procedure MENU_SCREEN(SCREEN_NUMBER:in out INTEGER;
        SCREEN_DATA:in out SCREEN_ARRAY);
  procedure GET_USER_INFO(USER_ID:in STRING;USER_ID_LENGTH:in
        INTEGER;CURRENT_TOPIC_NUMBER:out TOPICS;USER_NUMBER:out
INTEGER);
  procedure UPDATE_USER_INFO(CURRENT_TOPIC_NUMBER:in out TOPICS;
        USER_NUMBER:in INTEGER);
  procedure OPEN_SCREEN_DISPLAY(SCREEN_NUMBER:in out INTEGER;
        SCREEN_DATA:in out SCREEN_ARRAY);
  procedure SCREEN_DISPLAY(SCREEN_NUMBER:in out INTEGER;
        SCREEN_DATA:in out SCREEN_ARRAY;
        SCREEN_COUNT:in out INTEGER);

end SCREENS_PKG;

with
TEXT_IO,DIRECT_IO,EDITOR_PKG,ADA_LEX,ADA_LEX_IO,PARSER,DIAGNOS
IS;
use TEXT_IO,EDITOR_PKG,PARSER,ADA_LEX,ADA_LEX_IO,DIAGNOSIS;
package body SCREENS_PKG is

  subtype LINE_TYPE is STRING(1..80);
  package SCREEN_IO is new DIRECT_IO(LINE_TYPE);
  use SCREEN_IO;
  package USER_INOUT is new DIRECT_IO(USER_RECORD);
  use USER_INOUT;

  procedure QUIT_PROGRAM is
  begin
    null;
  end QUIT_PROGRAM;

  procedure LOAD_SCREENS(SCREEN_NUMBER:in INTEGER;
        SCREEN_DATA:in out SCREEN_ARRAY) is

    INFILE            :SCREEN_IO.FILE_TYPE;
    LINE              :STRING(1..80);
    FROM_NUMBER       :SCREEN_IO.POSITIVE_COUNT := 1;
    SCREEN_COUNTER    :INTEGER := 0;

  begin
    SCREEN_IO.OPEN(INFILE,MODE=>IN_FILE,NAME=>"screens.dat");
    SCREEN_COUNTER := SCREEN_NUMBER;
    while not END_OF_FILE(INFILE) loop
      for OUTER_INDEX in 1..14 loop
        if SCREEN_COUNTER = 1 then
          FROM_NUMBER := 1;
        else
          FROM_NUMBER :=
((SCREEN_IO.POSITIVE_COUNT(SCREEN_COUNTER)
```

```
                - 1) * 25) + 1;
      end if;
      for INNER_INDEX in 1..25 loop
        LINE := (others => ' ');
        SCREEN_IO.READ(INFILE,LINE,FROM_NUMBER);
        SCREEN_DATA(OUTER_INDEX,INNER_INDEX) := LINE;
        FROM_NUMBER := FROM_NUMBER + 1;
      end loop;
      SCREEN_COUNTER := SCREEN_COUNTER + 1;
    end loop;
  exit;
  end loop;
  SCREEN_IO.CLOSE(INFILE);
end LOAD_SCREENS;

procedure MENU_SCREEN(SCREEN_NUMBER:in out INTEGER;
     SCREEN_DATA:in out SCREEN_ARRAY) is

  subtype MENU_CHOICE is INTEGER range 0..12;
  package INTEGER_INOUT is new INTEGER_IO(MENU_CHOICE);
  use INTEGER_INOUT;

  CHOICE          :MENU_CHOICE;
  LINE1_24                :STRING(1..80);
  LINE_25         :STRING(1..80);

begin
  PUT(CHARACTER'VAL(12));
  for INDEX in 1..24 loop
    LINE1_24 := SCREEN_DATA(4,INDEX);
    PUT_LINE(LINE1_24(1..79));
  end loop;
  LINE_25 := SCREEN_DATA(4,25);
  PUT(LINE_25(1..78));
  INTEGER_INOUT.GET(CHOICE);
  case CHOICE is
    when 1 =>
      SCREEN_NUMBER := 5; -- Basics of Ada
    when 2 =>
      SCREEN_NUMBER := 13; -- Types
    when 3 =>
      SCREEN_NUMBER := 22; -- Expressions
    when 4 =>
      SCREEN_NUMBER := 28; -- Arrays and Strings
    when 5 =>
      SCREEN_NUMBER := 33; -- Input and Output
    when 6 =>
      SCREEN_NUMBER := 41; -- Control Statements
    when 7 =>
      SCREEN_NUMBER := 48; -- Subprograms
    when 8 =>
      SCREEN_NUMBER := 63; -- Records and Dynamic Structures
    when 9 =>
```

```
        SCREEN_NUMBER := 75; -- Exceptions
      when 10 =>
        SCREEN_NUMBER := 81; -- Files
      when 11 =>
        SCREEN_NUMBER := 89; -- Packages
      when 12 =>
        SCREEN_NUMBER := 103; -- Tasking
      when others =>
        USER_SPECIFIC := TRUE;
        SCREEN_NUMBER :=
TOPIC_SCREENS_ARRAY(CURRENT_TOPIC_NUMBER);
    end case;
    LOAD_SCREENS(SCREEN_NUMBER,SCREEN_DATA);
    SCREEN_COUNT := 1;
    SCREEN_DISPLAY(SCREEN_NUMBER,SCREEN_DATA,SCREEN_COUNT);

  exception
    when TEXT_IO.DATA_ERROR =>
      SKIP_LINE;
      SCREEN_NUMBER :=
TOPIC_SCREENS_ARRAY(CURRENT_TOPIC_NUMBER);
      LOAD_SCREENS(SCREEN_NUMBER,SCREEN_DATA);
      SCREEN_COUNT := 1;
      SCREEN_DISPLAY(SCREEN_NUMBER,SCREEN_DATA,SCREEN_COUNT);


  end MENU_SCREEN;

  procedure GET_USER_INFO(USER_ID:in STRING; USER_ID_LENGTH:in
INTEGER;
         CURRENT_TOPIC_NUMBER:out TOPICS;USER_NUMBER:out INTEGER)
is

    USER_FILE : USER_INOUT.FILE_TYPE;
    USER      : USER_RECORD;
    USER_REC_NO    : USER_INOUT.COUNT := 1;
    SUM       : INTEGER := 0;

  begin
    OPEN(USER_FILE,INOUT_FILE,"user.dat");
    loop
      READ(USER_FILE,USER,USER_REC_NO);
      if USER.USER_NAME = USER_ID then  -- found record
        CURRENT_TOPIC_NUMBER := USER.COMPLETED_TOPIC;
        exit;
      else
        USER_REC_NO := USER_REC_NO + 1;
      end if;
    end loop;
    USER_NUMBER := INTEGER(USER_REC_NO);
    CLOSE(USER_FILE);

  exception
```

```
    when USER_INOUT.NAME_ERROR => -- no file by that name
      CREATE(USER_FILE,INOUT_FILE,"user.dat");
      USER_REC_NO := 1;
      USER.USER_NAME(1..USER_ID_LENGTH) :=
USER_ID(1..USER_ID_LENGTH);
      WRITE(USER_FILE,USER,USER_REC_NO);
      CURRENT_TOPIC_NUMBER := 1;
      USER_NUMBER := INTEGER(USER_REC_NO);
      CLOSE(USER_FILE);
    when USER_INOUT.END_ERROR => -- user is not in user file
      USER_REC_NO := USER_REC_NO + 1;
      USER.USER_NAME(1..USER_ID_LENGTH) :=
USER_ID(1..USER_ID_LENGTH);
      WRITE(USER_FILE,USER,USER_REC_NO);
      CURRENT_TOPIC_NUMBER := 1;
      USER_NUMBER := INTEGER(USER_REC_NO);
      CLOSE(USER_FILE);
  end GET_USER_INFO;

  procedure UPDATE_USER_INFO(CURRENT_TOPIC_NUMBER:in out TOPICS;
        USER_NUMBER:in INTEGER)is

    USER_FILE : USER_INOUT.FILE_TYPE;
    USER      : USER_RECORD;
    USER_REC_NO    : USER_INOUT.COUNT := 1;

  begin
    OPEN(USER_FILE,OUT_FILE,"user.dat");
    USER_REC_NO := USER_INOUT.COUNT(USER_NUMBER);
    USER.USER_NAME := USER_ID;
    CURRENT_TOPIC_NUMBER := CURRENT_TOPIC_NUMBER + 1;
    USER.COMPLETED_TOPIC := CURRENT_TOPIC_NUMBER;
    WRITE(USER_FILE,USER,USER_REC_NO);
    CLOSE(USER_FILE);
  end UPDATE_USER_INFO;

  procedure OPEN_SCREEN_DISPLAY(SCREEN_NUMBER:in out INTEGER;
        SCREEN_DATA:in out SCREEN_ARRAY) is

    PAGE_CHOICE             :STRING(1..1);
    PAGE_CHOICE_LENGTH        :INTEGER := 0;
    CHOICE                  :CHARACTER;
    LINE1_24                :STRING(1..80);
    LINE_25                 :STRING(1..80);

  begin
    -- Opening Screen
    SCREEN_NUMBER := 1;
    PUT(CHARACTER'VAL(12));
    for INDEX in 1..24 loop
      LINE1_24 := SCREEN_DATA(1,INDEX);
      PUT_LINE(LINE1_24(1..79));
    end loop;
```

```
    LINE_25 := SCREEN_DATA(1,25);
    PUT(LINE_25(1..33));
    GET_LINE(USER_ID,USER_ID_LENGTH);


GET_USER_INFO(USER_ID,USER_ID_LENGTH,CURRENT_TOPIC_NUMBER,US
ER_NUMBER);
    -- Screen Manipulation Information
    SCREEN_NUMBER := 2;
    PUT(CHARACTER'VAL(12));
    for INDEX in 1..24 loop
      LINE1_24 := SCREEN_DATA(2,INDEX);
      PUT_LINE(LINE1_24(1..79));
    end loop;
    LINE_25 := SCREEN_DATA(2,25);
    PUT(LINE_25(1..78));
    GET_line(PAGE_CHOICE,page_choice_length);
    CHOICE := PAGE_CHOICE(1);
    case CHOICE is
      when 'p' | 'P' =>
        SCREEN_COUNT := 1;

SCREEN_DISPLAY(SCREEN_NUMBER,SCREEN_DATA,SCREEN_COUNT);
      when 'Q' | 'q' =>
        QUIT_PROGRAM;
      when others =>
        null;  -- fall thru

    end case;
    -- Editor Information
    SCREEN_NUMBER := 3;
    PUT(CHARACTER'VAL(12));
    for INDEX in 1..24 loop
      LINE1_24 := SCREEN_DATA(3,INDEX);
      PUT_LINE(LINE1_24(1..79));
    end loop;
    LINE_25 := SCREEN_DATA(3,25);
    PUT(LINE_25(1..78));
    skip_line;
    GET_LINE(PAGE_CHOICE,PAGE_CHOICE_LENGTH);
    CHOICE := PAGE_CHOICE(1);
    case CHOICE is
      when 'p' | 'P' =>
        SCREEN_NUMBER := SCREEN_NUMBER - 1;
        SCREEN_COUNT := 2;

SCREEN_DISPLAY(SCREEN_NUMBER,SCREEN_DATA,SCREEN_COUNT);
      when 'Q' | 'q' =>
        QUIT_PROGRAM;
      when others =>
        MENU_SCREEN(SCREEN_NUMBER,SCREEN_DATA);
    end case;
  end OPEN_SCREEN_DISPLAY;
```

```
procedure SCREEN_DISPLAY(SCREEN_NUMBER:in out INTEGER;
      SCREEN_DATA:in out SCREEN_ARRAY;
      SCREEN_COUNT:in out INTEGER) is

  PAGE_CHOICE          :STRING(1..1);
  PAGE_CHOICE_LENGTH      :INTEGER := 0;
  CHOICE               :CHARACTER;
  LINE1_24                   :STRING(1..80);
  LINE_25              :STRING(1..80);
  MAX_NUMBER                 :INTEGER := 112;

  procedure SELECT_PROBLEMS is

    type PROBLEM_RECORD is
      record
        PROBLEM_LINE : STRING(1..78);
    end record;

    package PROBLEM_INOUT is new DIRECT_IO(PROBLEM_RECORD);
    use PROBLEM_INOUT;

    BAD              : BOOLEAN := FALSE;
    SOLVED             : BOOLEAN := FALSE;
    EMPTY_BUFFER : STRING(1..78) := (others => ' ');
    PROBLEM_LINE_BUFFER : STRING(1..78) := EMPTY_BUFFER;
    PROBLEM_LINE_LENGTH : INTEGER;
    PROBLEM_FILE      : PROBLEM_INOUT.FILE_TYPE;
    SOLUTION_FILE     : PROBLEM_INOUT.FILE_TYPE;
    STUDENT_FILE      : TEXT_IO.FILE_TYPE;
    LINE            : STRING(1..78);
    EMPTY_LINE        : STRING(1..78) := (others => ' ');
    EXPERT_FILE       : TEXT_IO.FILE_TYPE;
    PROBLEM          : PROBLEM_RECORD;
    REC_NO          : PROBLEM_INOUT.COUNT;
    PROBLEM_NUMBER     : PROBLEM_TYPE := 1;
    START_STATE       : INTEGER;
    PROB_START        : INTEGER;
    PROB_STOP        : INTEGER;
    SOLN_START        : INTEGER;
    SOLN_STOP        : INTEGER;

    procedure INCREMENT(PROBLEM_NUMBER:in out PROBLEM_TYPE) is

    begin
      PROBLEM_NUMBER := PROBLEM_NUMBER + 1;
      if PROBLEM_NUMBER > 5 then
        PROBLEM_NUMBER := 1;
      end if;
      PUT(integer'image(PROBLEM_NUMBER));
    end INCREMENT;

procedure COPY_STUDENT_SOLN is
```

```
  EMPTY_LINE      : STRING(1..78) := (others => ' ');
  LINE            : STRING(1..78);
  INDEX           : NATURAL;
  DONE            : BOOLEAN := FALSE;


begin
  if IS_OPEN(STUDENT_FILE) then
    DELETE(STUDENT_FILE);
  end if;
  CREATE(STUDENT_FILE,NAME => "student.dat");
  for I in 1 .. editor_pkg.last_line loop -- for each line
   LINE(1..78) := EDITOR_PKG.LINES(I);
    DONE := FALSE;
    INDEX := 1;
    while not DONE loop
      if LINE(INDEX) = ASCII.CR then
        PUT(STUDENT_FILE,ASCII.LF);
        DONE := TRUE;
      else
        PUT(STUDENT_FILE,LINE(INDEX));
      end if;
      INDEX := INDEX + 1;
    end loop;
  end loop;
 end COPY_STUDENT_SOLN;


procedure COPY_EXPERT_SOLN is

  EMPTY_LINE      : STRING(1..80) := (others => ' ');
  LINE            : STRING(1..80);
  INDEX           : NATURAL;
  DONE            : BOOLEAN := FALSE;

begin
  if IS_OPEN(EXPERT_FILE) then
    DELETE(EXPERT_FILE);
  end if;
  CREATE(EXPERT_FILE,NAME => "expert.dat");
  put("OPENED expert.dat");
  for REC_NO in PROBLEM_INOUT.COUNT(SOLN_START) ..
      PROBLEM_INOUT.COUNT(SOLN_STOP) loop
    PROBLEM_INOUT.READ(SOLUTION_FILE,PROBLEM,REC_NO);
    LINE(1..78) := PROBLEM.PROBLEM_LINE;
    DONE := FALSE;
    INDEX := 1;
    while not DONE loop
      if LINE(INDEX) = ASCII.CR then
        PUT(EXPERT_FILE,ASCII.LF);
        DONE := TRUE;
      else
```

```
          PUT(EXPERT_FILE,LINE(INDEX));
        end if;
        INDEX := INDEX + 1;
      end loop;
    end loop;
 end COPY_EXPERT_SOLN;


procedure CHECK_SOLUTION(SOLVED:in out BOOLEAN) is

begin
   ADA_LEX_IO.OPEN_INPUT("student.dat");
   ADA_LEX_IO.CREATE_OUTPUT;PUT("parsing");
   YYPARSE;
   STUDENT_PTR := HW_PTR;
   ADA_LEX_IO.CLOSE_INPUT;
   ADA_LEX_IO.CLOSE_OUTPUT;
   SET_INPUT(STANDARD_INPUT);  -- due to changes made in
   SET_OUTPUT(STANDARD_OUTPUT); -- ada_lex_io package
   ADA_LEX.LINES := 1;
   ADA_LEX.VARIABLE := 0;put("now expert");
   COPY_EXPERT_SOLN;put("copied expert");
   ADA_LEX_IO.OPEN_INPUT("expert.dat");
   ADA_LEX_IO.CREATE_OUTPUT;
   YYPARSE;
   EXPERT_PTR := HW_PTR;
   ADA_LEX_IO.CLOSE_INPUT;
   ADA_LEX_IO.CLOSE_OUTPUT;
   SET_INPUT(STANDARD_INPUT);  -- due to changes made in
   SET_OUTPUT(STANDARD_OUTPUT); -- ada_lex_io package
   DIAGNOSE(STUDENT_PTR,EXPERT_PTR,SOLVED);
   DELETE(STUDENT_FILE);
   DELETE(EXPERT_FILE);
   if SOLVED then
     PUT("solved");
   else
     PUT("not solved");
   end if;
end CHECK_SOLUTION;

procedure PRESENT_PROBLEM(PROBLEM_NUMBER:in PROBLEM_TYPE;
  CURRENT_TOPIC_NUMBER:in TOPICS) is

  EMPTY_LINE      : STRING(1..78) := (others => ' ');
  LINE            : STRING(1..78);

begin
  START_STATE := ITS_ADA_DATA(CURRENT_TOPIC_NUMBER)
    (PROBLEM_NUMBER).START_STATE;
  PROB_START := ITS_ADA_DATA(CURRENT_TOPIC_NUMBER)
    (PROBLEM_NUMBER).PROB_START;
  PROB_STOP := ITS_ADA_DATA(CURRENT_TOPIC_NUMBER)
    (PROBLEM_NUMBER).PROB_STOP;
```

```
  SOLN_START := ITS_ADA_DATA(CURRENT_TOPIC_NUMBER)
    (PROBLEM_NUMBER).SOLN_START;
  SOLN_STOP := ITS_ADA_DATA(CURRENT_TOPIC_NUMBER)
    (PROBLEM_NUMBER).SOLN_STOP;
--  TEXT_IO.PUT(CHARACTER'VAL(12));
  for REC_NO in PROBLEM_INOUT.COUNT(PROB_START)
      .. PROBLEM_INOUT.COUNT(PROB_STOP) loop
    PROBLEM_INOUT.READ(PROBLEM_FILE,PROBLEM,REC_NO);
    PROBLEM_LINE_BUFFER  := PROBLEM.PROBLEM_LINE;
    PUT_LINE(PROBLEM_LINE_BUFFER(1..78));
  end loop;
  NEW_LINE(2); PUT("calls editor");
  EDITOR;
end PRESENT_PROBLEM;

procedure PRESENT_EXAMPLE(PROBLEM_NUMBER:in PROBLEM_TYPE;
  CURRENT_TOPIC_NUMBER:in TOPICS) is

   CHAR  : CHARACTER;

begin
  START_STATE := ITS_ADA_DATA(CURRENT_TOPIC_NUMBER)
    (PROBLEM_NUMBER).START_STATE;
  PROB_START := ITS_ADA_DATA(CURRENT_TOPIC_NUMBER)
    (PROBLEM_NUMBER).PROB_START;
  PROB_STOP := ITS_ADA_DATA(CURRENT_TOPIC_NUMBER)
    (PROBLEM_NUMBER).PROB_STOP;
  SOLN_START := ITS_ADA_DATA(CURRENT_TOPIC_NUMBER)
    (PROBLEM_NUMBER).SOLN_START;
  SOLN_STOP := ITS_ADA_DATA(CURRENT_TOPIC_NUMBER)
    (PROBLEM_NUMBER).SOLN_STOP;
--  TEXT_IO.PUT(CHARACTER'VAL(12));
  for REC_NO in PROBLEM_INOUT.COUNT(PROB_START)
      .. PROBLEM_INOUT.COUNT(PROB_STOP) loop
    READ(PROBLEM_FILE,PROBLEM,REC_NO);
    PROBLEM_LINE_BUFFER  := PROBLEM.PROBLEM_LINE;
    PUT_LINE(PROBLEM_LINE_BUFFER(1..78));
  end loop;
  NEW_LINE(2);
  for REC_NO in PROBLEM_INOUT.COUNT(SOLN_START)
      .. PROBLEM_INOUT.COUNT(SOLN_STOP) loop
    READ(SOLUTION_FILE,PROBLEM,REC_NO);
    PROBLEM_LINE_BUFFER  := PROBLEM.PROBLEM_LINE;
    PUT_LINE(PROBLEM_LINE_BUFFER(1..78));
  end loop;
end PRESENT_EXAMPLE;

procedure REMEDIATE(CURRENT_TOPIC_NUMBER:in TOPICS) is

begin
  PUT("remediate");
end REMEDIATE;
```

```
    begin -- SELECT PROBLEMS
      OPEN(PROBLEM_FILE,IN_FILE,"prob.dat");
      OPEN(SOLUTION_FILE,IN_FILE,"soln.dat");
      loop
        SOLVED := FALSE;

PRESENT_PROBLEM(PROBLEM_NUMBER,CURRENT_TOPIC_NUMBER); --a
        COPY_STUDENT_SOLN;
        CHECK_SOLUTION(SOLVED);
        if SOLVED then -- problem a
          BAD := FALSE;PUT("true");new_line;SOLVED := FALSE;
          INCREMENT(PROBLEM_NUMBER);

PRESENT_PROBLEM(PROBLEM_NUMBER,CURRENT_TOPIC_NUMBER); -- b
          COPY_STUDENT_SOLN;
          CHECK_SOLUTION(SOLVED);
          if SOLVED then -- problem b
            exit; -- move on
          else -- not SOLVED, problem b
   PUT("false");SOLVED := FALSE;  INCREMENT(PROBLEM_NUMBER);

PRESENT_EXAMPLE(PROBLEM_NUMBER,CURRENT_TOPIC_NUMBER);
            INCREMENT(PROBLEM_NUMBER);
          end if;
        else -- not SOLVED, problem a
          if BAD then -- 2 BADs
            REMEDIATE(CURRENT_TOPIC_NUMBER);
            exit;
          else
            BAD := TRUE;PUT_line("bad");SOLVED := FALSE;
          end if;
          INCREMENT(PROBLEM_NUMBER);put("example");

PRESENT_EXAMPLE(PROBLEM_NUMBER,CURRENT_TOPIC_NUMBER);
          INCREMENT(PROBLEM_NUMBER);
        end if; PUT("end loop");
      end loop;
      CLOSE(PROBLEM_FILE);
      CLOSE(SOLUTION_FILE);
    end SELECT_PROBLEMS;

  begin

   SELECT_PROBLEMS;
  end SCREEN_DISPLAY;

end SCREENS_PKG;
```

```
-- MAIN.A
--
with SCREENS_PKG;
use SCREENS_PKG;

procedure MAIN is
  SCREEN_NUMBER : INTEGER := 1;

begin
        LOAD_SCREENS(SCREEN_NUMBER,SCREEN_DATA);
        OPEN_SCREEN_DISPLAY(SCREEN_NUMBER,SCREEN_DATA);
end MAIN;
```

REFERENCES

Anderson, John R., The Expert Module, in Polson, Martha and Richardson, Jeffery J., eds, *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Associates, 1988.

Bonar, Jeffery G. and Cunningham Robert, Bridge: Tutoring the Programming Process, in Pstoka, Joeseph, Massey, L. Dan, and Mutter, Sharon A., eds, *Intelligent Tutoring Systems: Lessons Learned,* Lawrence Erlbaum Associates, 1988.

Brown, J. S., Burton, R. R. and DeKleer, J., Pedagogical, Natural Language and Knowledge Engineering Techniques in SOPHIE I, II and III, in Sleeman, D. and Brown, J. S., eds, *Intelligent Tutoring Systems*, Acedemic Press, 1982.

Burns, Hugh and Capps, Charles, Foundations of Intelligent Tutoring Systems: An Introduction, in Polson, Martha and Richardson, Jeffery J., eds, *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Associates, 1988.

Burton, Richard R., The Environment Module of Intelligent Tutoring Systems, in Polson, Martha and Richardson, Jeffery J., eds, *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Associates, 1988.

Clancey, W. J., Tutoring Rules for Guiding a Case Method Dialogue, in Sleeman, D. and Brown, J. S., eds, *Intelligent Tutoring Systems*, Academic Press, 1982.

Department of Defense, *Reference Manual for the Ada Programming Language,* ANSI/MIL-STD-1815A, Government Printing Office, Washington, DC, 1983.

Department of Defense, *Requirement for High Order Programming Languages,* STRAWMAN, February 1975.

Department of Defense, *Requirement for High Order Programming Languages,* WOODENMAN, November 1975.

Department of Defense, *Requirement for High Order Programming Languages,* TINMAN, June 1976.

Department of Defense, *Requirement for High Order Programming Languages,* IRONMAN, January 1978.

Department of Defense, *Requirement for High Order Programming Languages,* STEELMAN, June 1978.

Department of Defense, *DoD Software Initiatives*, February 1983.

Frye, Douglas, Littman, David C., and Soloway, Elliot, The Next Wave of Problems in ITS: Confronting the "User Issues" of Interface Design and System Evaluation, in Pstoka, Joeseph, Massey, L. Dan, and Mutter, Sharon A., eds, *Intelligent Tutoring Systems: Lessons Learned,* Lawrence Erlbaum Associates, 1988.

Herro, John J., ADA_TUTR User's Manual, 1988.

Johnson, William B. , Pragmatic Considerations In Research, Development and Implementation of Intelligent Tutoring Systems, in Polson, Martha and Richardson, Jeffery J., eds, *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Associates, 1988.

Miller, James R., The Role of Human-Computer Interaction in Intelligent Tutoring Systems, in Polson, Martha and Richardson, Jeffery J., eds, *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Associates, 1988.

Nissen, John and Wallis, Peter, *Portability and Style in Ada,* Cambridge University Press, 1984.

Pirolli, Peter L. and Greeno, James G., The Problem Space of Instructional Design, in Pstoka, Joeseph, Massey, L. Dan, and Mutter, Sharon A., eds, *Intelligent Tutoring Systems: Lessons Learned,* Lawrence Erlbaum Associates, 1988.

Riehle, Richard, "ADA ,A Software Engineering Tool," *Interact*, pp 83, April 1989.

Sammet, Jean E., "Why Ada Is Not Just Another Programming Language," *Communications of the ACM*, pp 723-727, August 1986.

Schmalz, Ronald J., *AYACC User's Manual,* Arcadia Environment Research Project, Department of Information and Computer Science, University of California, Irvine, 1988.

Skansholm, Jan, *Ada From the Beginning*, Addison-Wesley, 1988.

Smith, R. L., and others, Computer-assisted Axiomatic Mathematics: Informal Rigor, in Lecarne, O. and Lewis, R., *Computer Education*, North Holland, Amersterdam, 1975.

Suppes, P., "Some Theoretical Models for Mathematics Learning," *Journal of Research and Development in Education,* v. 1, pp. 5-22, 1967.

Uhr, L, "Teaching Machine Programs that Generate Problems as a Function of Interaction with Students," *Proceedings of the 24th National Conference,* pp. 125-134, 1969.

VanLehn, Kurt, Student Modeling, in Polson, Martha and Richardson, Jeffery J., eds, *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Associates, 1988.

VanLehn, Kurt, Keynote Speaker, 1991 Conference on Intelligent Computer-Aided Training, Johnson Space Center, Houston, Texas, 21 November 1991.

Woods, P. and Hartley, J. R., "Some Learning Models for Arithmetic Tasks and Their Use in Computer-based Learning," *British Journal of Educational Psychology*, v. 41, pp. 35-48, 1971.

Woolf, B., Intelligent Tutoring Systems, A Survey, in *Exploring Artificial Intelligence,* Shrobe, Howard E., ed, Morgan Kaufmann Publishers, 1988.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Dr. Yuh-jeng Lee Naval Postgraduate School Code CS, Department of Computer Science Monterey, California 93943-5100	8
4.	LT Lori L. DeLooze Naval Space Command Dahlgren, Virginia 22448	1
5.	LCDR Leigh W.Bradbury Naval Postgraduate School Code CS, Department of Computer Science Monterey, California 93943-5100	1
6.	Chairman Code CS, Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	1

847-313

Thesis
D29815 DeLooze
c.1 ITS Ada.

DUDLEY KNOX LIBRARY
3 2768 00034226 5